The Swift Programming Language

Swift 5.5 Edition

Welcome to Swift

Converted by Evan at Apps Dissected -www.appsdissected.com

About Swift

Swift is a fantastic way to write software, whether it's for phones, desktops, servers, or anything else that runs code. It's a safe, fast, and interactive programming language that combines the best in modern language thinking with wisdom from the wider Apple engineering culture and the diverse contributions from its open-source community. The compiler is optimized for performance and the language is optimized for development, without compromising on either.

Swift is friendly to new programmers. It's an industrial-quality programming language that's as expressive and enjoyable as a scripting language. Writing Swift code in a playground lets you experiment with code and see the results immediately, without the overhead of building and running an app.

Swift defines away large classes of common programming errors by adopting modern programming patterns:

- Variables are always initialized before use.
- Array indices are checked for out-of-bounds errors.
- Integers are checked for overflow.
- Optionals ensure that `nil` values are handled explicitly.
- Memory is managed automatically.
- Error handling allows controlled recovery from unexpected failures.

Swift code is compiled and optimized to get the most out of modern hardware. The syntax and standard library have been designed based on the guiding principle that the obvious way to write your code should also perform the best. Its combination of safety and speed make Swift an

excellent choice for everything from “Hello, world!” to an entire operating system.

Swift combines powerful type inference and pattern matching with a modern, lightweight syntax, allowing complex ideas to be expressed in a clear and concise manner. As a result, code is not just easier to write, but easier to read and maintain as well.

Swift has been years in the making, and it continues to evolve with new features and capabilities. Our goals for Swift are ambitious. We can’t wait to see what you create with it.

Converted by Evan at Apps Dissected -www.appsdissected.com

Version Compatibility

This book describes Swift 5.5, the default version of Swift that's included in Xcode 13. You can use Xcode 13 to build targets that are written in either Swift 5.5, Swift 4.2, or Swift 4.

When you use Xcode 13 to build Swift 4 and Swift 4.2 code, most Swift 5.5 functionality is available. That said, the following changes are available only to code that uses Swift 5.5 or later:

- Functions that return an opaque type require the Swift 5.1 runtime.
- The `try?` expression doesn't introduce an extra level of optionality to expressions that already return optionals.
- Large integer literal initialization expressions are inferred to be of the correct integer type. For example, `UInt64(0xffff_ffff_ffff_ffff)` evaluates to the correct value rather than overflowing.

Concurrency requires Swift 5.5 or later, and a version of the Swift standard library that provides the corresponding concurrency types. On Apple platforms, set a deployment target of at least iOS 15, macOS 12, tvOS 15, or watchOS 8.0.

A target written in Swift 5.5 can depend on a target that's written in Swift 4.2 or Swift 4, and vice versa. This means, if you have a large project that's divided into multiple frameworks, you can migrate your code from Swift 4 to Swift 5.5 one framework at a time.

Converted by Evan at Apps Dissected -www.appsdissected.com

A Swift Tour

Tradition suggests that the first program in a new language should print the words “Hello, world!” on the screen. In Swift, this can be done in a single line:

```
print("Hello, world!")
// Prints "Hello, world!"
```

If you have written code in C or Objective-C, this syntax looks familiar to you—in Swift, this line of code is a complete program. You don’t need to import a separate library for functionality like input/output or string handling. Code written at global scope is used as the entry point for the program, so you don’t need a `main()` function. You also don’t need to write semicolons at the end of every statement.

This tour gives you enough information to start writing code in Swift by showing you how to accomplish a variety of programming tasks. Don’t worry if you don’t understand something—everything introduced in this tour is explained in detail in the rest of this book.

> NOTE
>
> On a Mac with Xcode installed, or on an iPad with Swift Playgrounds, you can open this chapter as a playground. Playgrounds allow you to edit the code listings and see the result immediately.
>
> **Download Playground**

Simple Values

Use `let` to make a constant and `var` to make a variable. The value of a constant doesn't need to be known at compile time, but you must assign it a value exactly once. This means you can use constants to name a value that you determine once but use in many places.

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

A constant or variable must have the same type as the value you want to assign to it. However, you don't always have to write the type explicitly. Providing a value when you create a constant or variable lets the compiler infer its type. In the example above, the compiler infers that `myVariable` is an integer because its initial value is an integer.

If the initial value doesn't provide enough information (or if isn't an initial value), specify the type by writing it after the variable, separated by a colon.

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

> EXPERIMENT
>
> Create a constant with an explicit type of `Float` and a value of `4`.

Values are never implicitly converted to another type. If you need to convert a value to a different type, explicitly make an instance of the desired type.

```
let label = "The width is "
let width = 94
let widthLabel = label + String(width)
```

> EXPERIMENT
>
> Try removing the conversion to `String` from the last line. What error do you get?

There's an even simpler way to include values in strings: Write the value in parentheses, and write a backslash (`\`) before the parentheses. For example:

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges)
     pieces of fruit."
```

> EXPERIMENT
>
> Use `\()` to include a floating-point calculation in a string and to include someone's name in a greeting.

Use three double quotation marks (`"""`) for strings that take up multiple lines. Indentation at the start of each quoted line is removed, as long as it matches the indentation of the closing quotation marks. For example:

```
let quotation = """
I said "I have \(apples) apples."
And then I said "I have \(apples + oranges) pieces
     of fruit."
"""
```

Create arrays and dictionaries using brackets (`[]`), and access their elements by writing the index or key in brackets. A comma is allowed after the last element.

```
var shoppingList = ["catfish", "water", "tulips"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

Arrays automatically grow as you add elements.

```
shoppingList.append("blue paint")
print(shoppingList)
```

To create an empty array or dictionary, use the initializer syntax.

```
let emptyArray: [String] = []
let emptyDictionary: [String: Float] = [:]
```

If type information can be inferred, you can write an empty array as `[]` and an empty dictionary as `[:]` — for example, when you set a new value for a variable or pass an argument to a function.

```
shoppingList = []
occupations = [:]
```

Control Flow

Use `if` and `switch` to make conditionals, and use `for-in`, `while`, and `repeat-while` to make loops. Parentheses around the condition or loop variable are optional. Braces around the body are required.

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
print(teamScore)
// Prints "11"
```

In an `if` statement, the conditional must be a Boolean expression—this means that code such as `if score { ... }` is an error, not an implicit comparison to zero.

You can use `if` and `let` together to work with values that might be missing. These values are represented as optionals. An optional value either contains a value or contains `nil` to indicate that a value is missing. Write a question mark (`?`) after the type of a value to mark the value as optional.

```
var optionalString: String? = "Hello"
print(optionalString == nil)
// Prints "false"

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

> EXPERIMENT
>
> Change `optionalName` to `nil`. What greeting do you get? Add an `else` clause that sets a different greeting if `optionalName` is `nil`.

If the optional value is `nil`, the conditional is `false` and the code in braces is skipped. Otherwise, the optional value is unwrapped and assigned to the constant after `let`, which makes the unwrapped value available inside the block of code.

Another way to handle optional values is to provide a default value using the `??` operator. If the optional value is missing, the default value is used instead.

```
let nickname: String? = nil
let fullName: String = "John Appleseed"
let informalGreeting = "Hi \(nickname ?? fullName)"
```

Switches support any kind of data and a wide variety of comparison operations—they aren't limited to integers and tests for equality.

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    print("Add some raisins and make ants on a log.")
case "cucumber", "watercress":
    print("That would make a good tea sandwich.")
case let x where x.hasSuffix("pepper"):
    print("Is it a spicy \(x)?")
default:
    print("Everything tastes good in soup.")
}
// Prints "Is it a spicy red pepper?"
```

> EXPERIMENT
>
> Try removing the default case. What error do you get?

Notice how `let` can be used in a pattern to assign the value that matched the pattern to a constant.

After executing the code inside the switch case that matched, the program exits from the switch statement. Execution doesn't continue to the next case, so you don't need to explicitly break out of the switch at the end of each case's code.

You use `for-in` to iterate over items in a dictionary by providing a pair of names to use for each key-value pair. Dictionaries are an unordered collection, so their keys and values are iterated over in an arbitrary order.

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (_, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
print(largest)
// Prints "25"
```

> EXPERIMENT
>
> Replace the _ with a variable name, and keep track of which kind of number was the largest.

Use `while` to repeat a block of code until a condition changes. The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

```
var n = 2
while n < 100 {
    n *= 2
}
print(n)
// Prints "128"

var m = 2
repeat {
    m *= 2
} while m < 100
print(m)
// Prints "128"
```

You can keep an index in a loop by using `..<` to make a range of indexes.

```
var total = 0
for i in 0..<4 {
    total += i
}
print(total)
// Prints "6"
```

Use `..<` to make a range that omits its upper value, and use `...` to make a range that includes both values.

Functions and Closures

Use `func` to declare a function. Call a function by following its name with a list of arguments in parentheses. Use `->` to separate the parameter names and types from the function's return type.

```
func greet(person: String, day: String) -> String {
    return "Hello \(person), today is \(day)."
}
greet(person: "Bob", day: "Tuesday")
```

> EXPERIMENT
>
> Remove the `day` parameter. Add a parameter to include today's lunch special in the greeting.

By default, functions use their parameter names as labels for their arguments. Write a custom argument label before the parameter name, or write _ to use no argument label.

```
func greet(_ person: String, on day: String) ->
     String {
    return "Hello \(person), today is \(day)."
}
greet("John", on: "Wednesday")
```

Use a tuple to make a compound value—for example, to return multiple values from a function. The elements of a tuple can be referred to either by name or by number.

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int) {
    var min = scores[0]
    var max = scores[0]
    var sum = 0

    for score in scores {
        if score > max {
            max = score
        } else if score < min {
            min = score
        }
        sum += score
    }

    return (min, max, sum)
}
let statistics = calculateStatistics(scores: [5, 3, 100, 3, 9])
print(statistics.sum)
// Prints "120"
print(statistics.2)
// Prints "120"
```

Functions can be nested. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that's long or complex.

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

Functions are a first-class type. This means that a function can return another function as its value.

```
func makeIncrementer() -> ((Int) -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

A function can take another function as one of its arguments.

```
func hasAnyMatches(list: [Int], condition: (Int) ->
     Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(list: numbers, condition: lessThanTen)
```

Functions are actually a special case of closures: blocks of code that can be called later. The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it's executed—you saw an example of this already with nested functions. You can write a closure without a name by surrounding code with braces (`{}`). Use `in` to separate the arguments and return type from the body.

```
numbers.map({ (number: Int) -> Int in
    let result = 3 * number
    return result
})
```

> EXPERIMENT
>
> Rewrite the closure to return zero for all odd numbers.

You have several options for writing closures more concisely. When a closure's type is already known, such as the callback for a delegate, you can omit the type of its parameters, its return type, or both. Single statement closures implicitly return the value of their only statement.

```
let mappedNumbers = numbers.map({ number in 3 *
     number })
print(mappedNumbers)
// Prints "[60, 57, 21, 36]"
```

You can refer to parameters by number instead of by name—this approach is especially useful in very short closures. A closure passed as the last argument to a function can appear immediately after the parentheses. When a closure is the only argument to a function, you can omit the parentheses entirely.

```
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
// Prints "[20, 19, 12, 7]"
```

Objects and Classes

Use `class` followed by the class's name to create a class. A property declaration in a class is written the same way as a constant or variable declaration, except that it's in the context of a class. Likewise, method and function declarations are written the same way.

```
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

> EXPERIMENT
>
> Add a constant property with `let`, and add another method that takes an argument.

Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

This version of the `Shape` class is missing something important: an initializer to set up the class when an instance is created. Use `init` to create one.

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

Notice how `self` is used to distinguish the `name` property from the `name` argument to the initializer. The arguments to the initializer are passed like a function call when you create an instance of the class. Every property needs a value assigned—either in its declaration (as with `numberOfSides`) or in the initializer (as with `name`).

Use `deinit` to create a deinitializer if you need to perform some cleanup before the object is deallocated.

Subclasses include their superclass name after their class name, separated by a colon. There's no requirement for classes to subclass any standard root class, so you can include or omit a superclass as needed.

Methods on a subclass that override the superclass's implementation are marked with `override`—overriding a method by accident, without `override`, is detected by the compiler as an error. The compiler also detects

methods with `override` that don't actually override any method in the superclass.

```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() -> Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
```

> EXPERIMENT
>
> Make another subclass of `NamedShape` called `Circle` that takes a radius and a name as arguments to its initializer. Implement an `area()` and a `simpleDescription()` method on the `Circle` class.

In addition to simple properties that are stored, properties can have a getter and a setter.

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }

    var perimeter: Double {
        get {
            return 3.0 * sideLength
        }
        set {
            sideLength = newValue / 3.0
        }
    }

    override func simpleDescription() -> String {
        return "An equilateral triangle with sides of length \(sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
print(triangle.perimeter)
```

```
// Prints "9.3"
triangle.perimeter = 9.9
print(triangle.sideLength)
// Prints "3.3000000000000003"
```

In the setter for `perimeter`, the new value has the implicit name `newValue`. You can provide an explicit name in parentheses after `set`.

Notice that the initializer for the `EquilateralTriangle` class has three different steps:

1. Setting the value of properties that the subclass declares.
2. Calling the superclass's initializer.
3. Changing the value of properties defined by the superclass. Any additional setup work that uses methods, getters, or setters can also be done at this point.

If you don't need to compute the property but still need to provide code that's run before and after setting a new value, use `willSet` and `didSet`. The code you provide is run any time the value changes outside of an initializer. For example, the class below ensures that the side length of its triangle is always the same as the side length of its square.

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
        willSet {
            square.sideLength = newValue.sideLength
        }
    }
    var square: Square {
        willSet {
            triangle.sideLength = newValue.sideLength
        }
    }
    init(size: Double, name: String) {
        square = Square(sideLength: size, name: name)
        triangle = EquilateralTriangle(sideLength: size, name: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
print(triangleAndSquare.square.sideLength)
// Prints "10.0"
print(triangleAndSquare.triangle.sideLength)
// Prints "10.0"
```

```
triangleAndSquare.square = Square(sideLength: 50,
     name: "larger square")
print(triangleAndSquare.triangle.sideLength)
// Prints "50.0"
```

When working with optional values, you can write `?` before operations like methods, properties, and subscripting. If the value before the `?` is `nil`, everything after the `?` is ignored and the value of the whole expression is `nil`. Otherwise, the optional value is unwrapped, and everything after the `?` acts on the unwrapped value. In both cases, the value of the whole expression is an optional value.

```
let optionalSquare: Square? = Square(sideLength:
     2.5, name: "optional square")
let sideLength = optionalSquare?.sideLength
```

Enumerations and Structures

Use `enum` to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

```
enum Rank: Int {
    case ace = 1
    case two, three, four, five, six, seven, eight, nine, ten
    case jack, queen, king

    func simpleDescription() -> String {
        switch self {
        case .ace:
            return "ace"
        case .jack:
            return "jack"
        case .queen:
            return "queen"
        case .king:
            return "king"
        default:
            return String(self.rawValue)
        }
    }
}
let ace = Rank.ace
let aceRawValue = ace.rawValue
```

> EXPERIMENT
>
> Write a function that compares two `Rank` values by comparing their raw values.

By default, Swift assigns the raw values starting at zero and incrementing by one each time, but you can change this behavior by explicitly specifying values. In the example above, `Ace` is explicitly given a raw value of `1`, and the rest of the raw values are assigned in order. You can also use strings or floating-point numbers as the raw type of an enumeration. Use the `rawValue` property to access the raw value of an enumeration case.

Use the `init?(rawValue:)` initializer to make an instance of an enumeration from a raw value. It returns either the enumeration case matching the raw value or `nil` if there's no matching `Rank`.

```
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription =
     convertedRank.simpleDescription()
}
```

The case values of an enumeration are actual values, not just another way of writing their raw values. In fact, in cases where there isn't a meaningful raw value, you don't have to provide one.

```
enum Suit {
    case spades, hearts, diamonds, clubs

    func simpleDescription() -> String {
        switch self {
        case .spades:
            return "spades"
        case .hearts:
            return "hearts"
        case .diamonds:
            return "diamonds"
        case .clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.hearts
let heartsDescription = hearts.simpleDescription()
```

> EXPERIMENT
>
> Add a `color()` method to `Suit` that returns “black” for spades and clubs, and returns “red” for hearts and diamonds.

Notice the two ways that the `hearts` case of the enumeration is referred to above: When assigning a value to the `hearts` constant, the enumeration case `Suit.hearts` is referred to by its full name because the constant doesn’t have an explicit type specified. Inside the switch, the enumeration case is referred to by the abbreviated form `.hearts` because the value of

`self` is already known to be a suit. You can use the abbreviated form anytime the value's type is already known.

If an enumeration has raw values, those values are determined as part of the declaration, which means every instance of a particular enumeration case always has the same raw value. Another choice for enumeration cases is to have values associated with the case—these values are determined when you make the instance, and they can be different for each instance of an enumeration case. You can think of the associated values as behaving like stored properties of the enumeration case instance. For example, consider the case of requesting the sunrise and sunset times from a server. The server either responds with the requested information, or it responds with a description of what went wrong.

```swift
enum ServerResponse {
    case result(String, String)
    case failure(String)
}

let success = ServerResponse.result("6:00 am", "8:09 pm")
let failure = ServerResponse.failure("Out of cheese.")

switch success {
case let .result(sunrise, sunset):
    print("Sunrise is at \(sunrise) and sunset is at \(sunset).")
case let .failure(message):
    print("Failure...  \(message)")
}
// Prints "Sunrise is at 6:00 am and sunset is at 8:09 pm."
```

> EXPERIMENT
>
> Add a third case to `ServerResponse` and to the switch.

Notice how the sunrise and sunset times are extracted from the `ServerResponse` value as part of matching the value against the switch cases.

Use `struct` to create a structure. Structures support many of the same behaviors as classes, including methods and initializers. One of the most important differences between structures and classes is that structures are always copied when they're passed around in your code, but classes are passed by reference.

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}
let threeOfSpades = Card(rank: .three, suit: .spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

> EXPERIMENT
>
> Write a function that returns an array containing a full deck of cards, with one card of each combination of rank and suit.

Protocols and Extensions

Use `protocol` to declare a protocol.

```
1  protocol ExampleProtocol {
2      var simpleDescription: String { get }
3      mutating func adjust()
4  }
```

Classes, enumerations, and structs can all adopt protocols.

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

EXPERIMENT

Add another requirement to `ExampleProtocol`. What changes do you need to make to `SimpleClass` and `SimpleStructure` so that they still conform to the protocol?

Notice the use of the `mutating` keyword in the declaration of `SimpleStructure` to mark a method that modifies the structure. The declaration of `SimpleClass` doesn't need any of its methods marked as mutating because methods on a class can always modify the class.

Use `extension` to add functionality to an existing type, such as new methods and computed properties. You can use an extension to add protocol conformance to a type that's declared elsewhere, or even to a type that you imported from a library or framework.

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
        return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
print(7.simpleDescription)
// Prints "The number 7"
```

> EXPERIMENT
>
> Write an extension for the `Double` type that adds an `absoluteValue` property.

You can use a protocol name just like any other named type—for example, to create a collection of objects that have different types but that all conform to a single protocol. When you work with values whose type is a protocol type, methods outside the protocol definition aren't available.

```
let protocolValue: ExampleProtocol = a
print(protocolValue.simpleDescription)
// Prints "A very simple class.  Now 100% adjusted."
// print(protocolValue.anotherProperty)  //
     Uncomment to see the error
```

Even though the variable `protocolValue` has a runtime type of `SimpleClass`, the compiler treats it as the given type of `ExampleProtocol`. This means that you can't accidentally access methods or properties that the class implements in addition to its protocol conformance.

Error Handling

You represent errors using any type that adopts the `Error` protocol.

```
enum PrinterError: Error {
    case outOfPaper
    case noToner
    case onFire
}
```

Use `throw` to throw an error and `throws` to mark a function that can throw an error. If you throw an error in a function, the function returns immediately and the code that called the function handles the error.

```
func send(job: Int, toPrinter printerName: String) throws -> String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

There are several ways to handle errors. One way is to use `do-catch`. Inside the `do` block, you mark code that can throw an error by writing `try` in front of it. Inside the `catch` block, the error is automatically given the name `error` unless you give it a different name.

```
do {
    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")
    print(printerResponse)
} catch {
    print(error)
}
// Prints "Job sent"
```

> EXPERIMENT
>
> Change the printer name to `"Never Has Toner"`, so that the `send(job:toPrinter:)` function throws an error.

You can provide multiple `catch` blocks that handle specific errors. You write a pattern after `catch` just as you do after `case` in a switch.

```
do {
    let printerResponse = try send(job: 1440,
     toPrinter: "Gutenberg")
    print(printerResponse)
} catch PrinterError.onFire {
    print("I'll just put this over here, with the
     rest of the fire.")
} catch let printerError as PrinterError {
    print("Printer error: \(printerError).")
} catch {
    print(error)
}
// Prints "Job sent"
```

> EXPERIMENT
>
> Add code to throw an error inside the `do` block. What kind of error do you need to throw so that the error is handled by the first `catch` block? What about the second and third blocks?

Another way to handle errors is to use `try?` to convert the result to an optional. If the function throws an error, the specific error is discarded and the result is `nil`. Otherwise, the result is an optional containing the value that the function returned.

```
let printerSuccess = try? send(job: 1884, toPrinter:
     "Mergenthaler")
let printerFailure = try? send(job: 1885, toPrinter:
     "Never Has Toner")
```

Use `defer` to write a block of code that's executed after all other code in the function, just before the function returns. The code is executed regardless of whether the function throws an error. You can use `defer` to write setup and cleanup code next to each other, even though they need to be executed at different times.

```
var fridgeIsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]

func fridgeContains(_ food: String) -> Bool {
    fridgeIsOpen = true
    defer {
        fridgeIsOpen = false
    }

    let result = fridgeContent.contains(food)
    return result
}
fridgeContains("banana")
print(fridgeIsOpen)
// Prints "false"
```

Generics

Write a name inside angle brackets to make a generic function or type.

```swift
func makeArray<Item>(repeating item: Item,
     numberOfTimes: Int) -> [Item] {
    var result: [Item] = []
    for _ in 0..<numberOfTimes {
        result.append(item)
    }
    return result
}
makeArray(repeating: "knock", numberOfTimes: 4)
```

You can make generic forms of functions and methods, as well as classes, enumerations, and structures.

```swift
// Reimplement the Swift standard library's optional
     type
enum OptionalValue<Wrapped> {
    case none
    case some(Wrapped)
}
var possibleInteger: OptionalValue<Int> = .none
possibleInteger = .some(100)
```

Use `where` right before the body to specify a list of requirements—for example, to require the type to implement a protocol, to require two types to be the same, or to require a class to have a particular superclass.

```swift
func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> Bool
    where T.Element: Equatable, T.Element == U.Element
{
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
        }
    }
    return false
}
anyCommonElements([1, 2, 3], [3])
```

> EXPERIMENT
>
> Modify the `anyCommonElements(_:_:)` function to make a function that returns an array of the elements that any two sequences have in common.

Writing `<T: Equatable>` is the same as writing `<T> ... where T: Equatable`.

Language Guide

Converted by Evan at Apps Dissected -www.appsdissected.com

The Basics

Swift is a new programming language for iOS, macOS, watchOS, and tvOS app development. Nonetheless, many parts of Swift will be familiar from your experience of developing in C and Objective-C.

Swift provides its own versions of all fundamental C and Objective-C types, including `Int` for integers, `Double` and `Float` for floating-point values, `Bool` for Boolean values, and `String` for textual data. Swift also provides powerful versions of the three primary collection types, `Array`, `Set`, and `Dictionary`, as described in Collection Types.

Like C, Swift uses variables to store and refer to values by an identifying name. Swift also makes extensive use of variables whose values can't be changed. These are known as constants, and are much more powerful than constants in C. Constants are used throughout Swift to make code safer and clearer in intent when you work with values that don't need to change.

In addition to familiar types, Swift introduces advanced types not found in Objective-C, such as tuples. Tuples enable you to create and pass around groupings of values. You can use a tuple to return multiple values from a function as a single compound value.

Swift also introduces optional types, which handle the absence of a value. Optionals say either "there *is* a value, and it equals *x*" or "there *isn't* a value at all". Using optionals is similar to using `nil` with pointers in Objective-C, but they work for any type, not just classes. Not only are optionals safer and more expressive than `nil` pointers in Objective-C, they're at the heart of many of Swift's most powerful features.

Swift is a *type-safe* language, which means the language helps you to be clear about the types of values your code can work with. If part of your code requires a `String`, type safety prevents you from passing it an `Int` by mistake. Likewise, type safety prevents you from accidentally passing an optional `String` to a piece of code that requires a non-optional `String`.

Type safety helps you catch and fix errors as early as possible in the development process.

Constants and Variables

Constants and variables associate a name (such as `maximumNumberOfLoginAttempts` or `welcomeMessage`) with a value of a particular type (such as the number `10` or the string `"Hello"`). The value of a *constant* can't be changed once it's set, whereas a *variable* can be set to a different value in the future.

Declaring Constants and Variables

Constants and variables must be declared before they're used. You declare constants with the `let` keyword and variables with the `var` keyword. Here's an example of how constants and variables can be used to track the number of login attempts a user has made:

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

This code can be read as:

"Declare a new constant called `maximumNumberOfLoginAttempts`, and give it a value of `10`. Then, declare a new variable called `currentLoginAttempt`, and give it an initial value of `0`."

In this example, the maximum number of allowed login attempts is declared as a constant, because the maximum value never changes. The current login attempt counter is declared as a variable, because this value must be incremented after each failed login attempt.

You can declare multiple constants or multiple variables on a single line, separated by commas:

```
var x = 0.0, y = 0.0, z = 0.0
```

> NOTE
>
> If a stored value in your code won't change, always declare it as a constant with the `let` keyword. Use variables only for storing values that need to be able to change.

Type Annotations

You can provide a *type annotation* when you declare a constant or variable, to be clear about the kind of values the constant or variable can store. Write a type annotation by placing a colon after the constant or variable name, followed by a space, followed by the name of the type to use.

This example provides a type annotation for a variable called `welcomeMessage`, to indicate that the variable can store `String` values:

```
var welcomeMessage: String
```

The colon in the declaration means "…of type…," so the code above can be read as:

"Declare a variable called `welcomeMessage` that's of type `String`."

The phrase "of type `String`" means "can store any `String` value." Think of it as meaning "the type of thing" (or "the kind of thing") that can be stored.

The `welcomeMessage` variable can now be set to any string value without error:

```
welcomeMessage = "Hello"
```

You can define multiple related variables of the same type on a single line, separated by commas, with a single type annotation after the final variable name:

```
var red, green, blue: Double
```

> NOTE
>
> It's rare that you need to write type annotations in practice. If you provide an initial value for a constant or variable at the point that it's defined, Swift can almost always infer the type to be used for that constant or variable, as described in Type Safety and Type Inference. In the `welcomeMessage` example above, no initial value is provided, and so the type of the `welcomeMessage` variable is specified with a type annotation rather than being inferred from an initial value.

Naming Constants and Variables

Constant and variable names can contain almost any character, including Unicode characters:

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Constant and variable names can't contain whitespace characters, mathematical symbols, arrows, private-use Unicode scalar values, or line- and box-drawing characters. Nor can they begin with a number, although numbers may be included elsewhere within the name.

Once you've declared a constant or variable of a certain type, you can't declare it again with the same name, or change it to store values of a different type. Nor can you change a constant into a variable or a variable into a constant.

> NOTE
>
> If you need to give a constant or variable the same name as a reserved Swift keyword, surround the keyword with backticks (`) when using it as a name. However, avoid using keywords as names unless you have absolutely no choice.

You can change the value of an existing variable to another value of a compatible type. In this example, the value of `friendlyWelcome` is changed from `"Hello!"` to `"Bonjour!"`:

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome is now "Bonjour!"
```

Unlike a variable, the value of a constant can't be changed after it's set. Attempting to do so is reported as an error when your code is compiled:

```
let languageName = "Swift"
languageName = "Swift++"
// This is a compile-time error: languageName cannot
     be changed.
```

Printing Constants and Variables

You can print the current value of a constant or variable with the `print(_:separator:terminator:)` function:

```
print(friendlyWelcome)
// Prints "Bonjour!"
```

The `print(_:separator:terminator:)` function is a global function that prints one or more values to an appropriate output. In Xcode, for example,

the `print(_:separator:terminator:)` function prints its output in Xcode's "console" pane. The `separator` and `terminator` parameter have default values, so you can omit them when you call this function. By default, the function terminates the line it prints by adding a line break. To print a value without a line break after it, pass an empty string as the terminator—for example, `print(someValue, terminator: "")`. For information about parameters with default values, see Default Parameter Values.

Swift uses *string interpolation* to include the name of a constant or variable as a placeholder in a longer string, and to prompt Swift to replace it with the current value of that constant or variable. Wrap the name in parentheses and escape it with a backslash before the opening parenthesis:

```
print("The current value of friendlyWelcome is \(friendlyWelcome)")
// Prints "The current value of friendlyWelcome is Bonjour!"
```

> NOTE
> All options you can use with string interpolation are described in String Interpolation.

Comments

Use comments to include nonexecutable text in your code, as a note or reminder to yourself. Comments are ignored by the Swift compiler when your code is compiled.

Comments in Swift are very similar to comments in C. Single-line comments begin with two forward-slashes (`//`):

```
// This is a comment.
```

Multiline comments start with a forward-slash followed by an asterisk (`/*`) and end with an asterisk followed by a forward-slash (`*/`):

```
/* This is also a comment
but is written over multiple lines. */
```

Unlike multiline comments in C, multiline comments in Swift can be nested inside other multiline comments. You write nested comments by starting a multiline comment block and then starting a second multiline comment within the first block. The second block is then closed, followed by the first block:

```
/* This is the start of the first multiline comment.
 /* This is the second, nested multiline comment. */
This is the end of the first multiline comment. */
```

Nested multiline comments enable you to comment out large blocks of code quickly and easily, even if the code already contains multiline comments.

Semicolons

Unlike many other languages, Swift doesn't require you to write a semicolon (`;`) after each statement in your code, although you can do so if you wish. However, semicolons *are* required if you want to write multiple separate statements on a single line:

```
let cat = "🐱"; print(cat)
// Prints "🐱"
```

Integers

Integers are whole numbers with no fractional component, such as `42` and `-23`. Integers are either *signed* (positive, zero, or negative) or *unsigned* (positive or zero).

Swift provides signed and unsigned integers in 8, 16, 32, and 64 bit forms. These integers follow a naming convention similar to C, in that an 8-bit unsigned integer is of type `UInt8`, and a 32-bit signed integer is of type `Int32`. Like all types in Swift, these integer types have capitalized names.

Integer Bounds

You can access the minimum and maximum values of each integer type with its `min` and `max` properties:

```
let minValue = UInt8.min  // minValue is equal to 0,
     and is of type UInt8
let maxValue = UInt8.max  // maxValue is equal to
     255, and is of type UInt8
```

The values of these properties are of the appropriate-sized number type (such as `UInt8` in the example above) and can therefore be used in expressions alongside other values of the same type.

Int

In most cases, you don't need to pick a specific size of integer to use in your code. Swift provides an additional integer type, `Int`, which has the same size as the current platform's native word size:

- On a 32-bit platform, `Int` is the same size as `Int32`.
- On a 64-bit platform, `Int` is the same size as `Int64`.

Unless you need to work with a specific size of integer, always use `Int` for integer values in your code. This aids code consistency and interoperability. Even on 32-bit platforms, `Int` can store any value between `-2,147,483,648` and `2,147,483,647`, and is large enough for many integer ranges.

UInt

Swift also provides an unsigned integer type, `UInt`, which has the same size as the current platform's native word size:

- On a 32-bit platform, `UInt` is the same size as `UInt32`.
- On a 64-bit platform, `UInt` is the same size as `UInt64`.

> NOTE
>
> Use `UInt` only when you specifically need an unsigned integer type with the same size as the platform's native word size. If this isn't the case, `Int` is preferred, even when the values to be stored are known to be nonnegative. A consistent use of `Int` for integer values aids code interoperability, avoids the need to convert between different number types, and matches integer type inference, as described in Type Safety and Type Inference.

Floating-Point Numbers

Floating-point numbers are numbers with a fractional component, such as `3.14159`, `0.1`, and `-273.15`.

Floating-point types can represent a much wider range of values than integer types, and can store numbers that are much larger or smaller than can be stored in an `Int`. Swift provides two signed floating-point number types:

- `Double` represents a 64-bit floating-point number.
- `Float` represents a 32-bit floating-point number.

> NOTE
>
> `Double` has a precision of at least 15 decimal digits, whereas the precision of `Float` can be as little as 6 decimal digits. The appropriate floating-point type to use depends on the nature and range of values you need to work with in your code. In situations where either type would be appropriate, `Double` is preferred.

Type Safety and Type Inference

Swift is a *type-safe* language. A type safe language encourages you to be clear about the types of values your code can work with. If part of your code requires a `String`, you can't pass it an `Int` by mistake.

Because Swift is type safe, it performs *type checks* when compiling your code and flags any mismatched types as errors. This enables you to catch and fix errors as early as possible in the development process.

Type-checking helps you avoid errors when you're working with different types of values. However, this doesn't mean that you have to specify the type of every constant and variable that you declare. If you don't specify the type of value you need, Swift uses *type inference* to work out the appropriate type. Type inference enables a compiler to deduce the type of a

particular expression automatically when it compiles your code, simply by examining the values you provide.

Because of type inference, Swift requires far fewer type declarations than languages such as C or Objective-C. Constants and variables are still explicitly typed, but much of the work of specifying their type is done for you.

Type inference is particularly useful when you declare a constant or variable with an initial value. This is often done by assigning a *literal value* (or *literal*) to the constant or variable at the point that you declare it. (A literal value is a value that appears directly in your source code, such as `42` and `3.14159` in the examples below.)

For example, if you assign a literal value of `42` to a new constant without saying what type it is, Swift infers that you want the constant to be an `Int`, because you have initialized it with a number that looks like an integer:

```
let meaningOfLife = 42
// meaningOfLife is inferred to be of type Int
```

Likewise, if you don't specify a type for a floating-point literal, Swift infers that you want to create a `Double`:

```
let pi = 3.14159
// pi is inferred to be of type Double
```

Swift always chooses `Double` (rather than `Float`) when inferring the type of floating-point numbers.

If you combine integer and floating-point literals in an expression, a type of `Double` will be inferred from the context:

```
let anotherPi = 3 + 0.14159
// anotherPi is also inferred to be of type Double
```

The literal value of `3` has no explicit type in and of itself, and so an appropriate output type of `Double` is inferred from the presence of a floating-point literal as part of the addition.

Numeric Literals

Integer literals can be written as:

- A *decimal* number, with no prefix
- A *binary* number, with a `0b` prefix
- An *octal* number, with a `0o` prefix
- A *hexadecimal* number, with a `0x` prefix

All of these integer literals have a decimal value of `17`:

```
let decimalInteger = 17
let binaryInteger = 0b10001       // 17 in binary notation
let octalInteger = 0o21           // 17 in octal notation
let hexadecimalInteger = 0x11     // 17 in hexadecimal notation
```

Floating-point literals can be decimal (with no prefix), or hexadecimal (with a `0x` prefix). They must always have a number (or hexadecimal number) on both sides of the decimal point. Decimal floats can also have an optional *exponent*, indicated by an uppercase or lowercase `e`; hexadecimal floats must have an exponent, indicated by an uppercase or lowercase `p`.

For decimal numbers with an exponent of `exp`, the base number is multiplied by 10^{exp}:

- `1.25e2` means 1.25×10^{2}, or `125.0`.
- `1.25e-2` means 1.25×10^{-2}, or `0.0125`.

For hexadecimal numbers with an exponent of `exp`, the base number is multiplied by 2^{exp}:

- `0xFp2` means 15×2^{2}, or `60.0`.
- `0xFp-2` means 15×2^{-2}, or `3.75`.

All of these floating-point literals have a decimal value of `12.1875`:

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
let hexadecimalDouble = 0xC.3p0
```

Numeric literals can contain extra formatting to make them easier to read. Both integers and floats can be padded with extra zeros and can contain underscores to help with readability. Neither type of formatting affects the underlying value of the literal:

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
let justOverOneMillion = 1_000_000.000_000_1
```

Numeric Type Conversion

Use the `Int` type for all general-purpose integer constants and variables in your code, even if they're known to be nonnegative. Using the default integer type in everyday situations means that integer constants and variables are immediately interoperable in your code and will match the inferred type for integer literal values.

Use other integer types only when they're specifically needed for the task at hand, because of explicitly sized data from an external source, or for performance, memory usage, or other necessary optimization. Using explicitly sized types in these situations helps to catch any accidental value overflows and implicitly documents the nature of the data being used.

Integer Conversion

The range of numbers that can be stored in an integer constant or variable is different for each numeric type. An `Int8` constant or variable can store numbers between `-128` and `127`, whereas a `UInt8` constant or variable can store numbers between `0` and `255`. A number that won't fit into a constant or variable of a sized integer type is reported as an error when your code is compiled:

```
let cannotBeNegative: UInt8 = -1
// UInt8 can't store negative numbers, and so this will report an error
let tooBig: Int8 = Int8.max + 1
// Int8 can't store a number larger than its maximum value,
// and so this will also report an error
```

Because each numeric type can store a different range of values, you must opt in to numeric type conversion on a case-by-case basis. This opt-in approach prevents hidden conversion errors and helps make type conversion intentions explicit in your code.

To convert one specific number type to another, you initialize a new number of the desired type with the existing value. In the example below, the constant `twoThousand` is of type `UInt16`, whereas the constant `one` is of type `UInt8`. They can't be added together directly, because they're not of the same type. Instead, this example calls `UInt16(one)` to create a new `UInt16` initialized with the value of `one`, and uses this value in place of the original:

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
let twoThousandAndOne = twoThousand + UInt16(one)
```

Because both sides of the addition are now of type `UInt16`, the addition is allowed. The output constant (`twoThousandAndOne`) is inferred to be of type `UInt16`, because it's the sum of two `UInt16` values.

`SomeType(ofInitialValue)` is the default way to call the initializer of a Swift type and pass in an initial value. Behind the scenes, `UInt16` has an initializer that accepts a `UInt8` value, and so this initializer is used to make

a new `UInt16` from an existing `UInt8`. You can't pass in *any* type here, however—it has to be a type for which `UInt16` provides an initializer. Extending existing types to provide initializers that accept new types (including your own type definitions) is covered in Extensions.

Integer and Floating-Point Conversion

Conversions between integer and floating-point numeric types must be made explicit:

```
let three = 3
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// pi equals 3.14159, and is inferred to be of type
     Double
```

Here, the value of the constant `three` is used to create a new value of type `Double`, so that both sides of the addition are of the same type. Without this conversion in place, the addition would not be allowed.

Floating-point to integer conversion must also be made explicit. An integer type can be initialized with a `Double` or `Float` value:

```
let integerPi = Int(pi)
// integerPi equals 3, and is inferred to be of type
     Int
```

Floating-point values are always truncated when used to initialize a new integer value in this way. This means that `4.75` becomes `4`, and `-3.9` becomes `-3`.

> NOTE
>
> The rules for combining numeric constants and variables are different from the rules for numeric literals. The literal value `3` can be added directly to the literal value `0.14159`, because number literals don't have an explicit type in and of themselves. Their type is inferred only at the point that they're evaluated by the compiler.

Type Aliases

Type aliases define an alternative name for an existing type. You define type aliases with the `typealias` keyword.

Type aliases are useful when you want to refer to an existing type by a name that's contextually more appropriate, such as when working with data of a specific size from an external source:

```
typealias AudioSample = UInt16
```

Once you define a type alias, you can use the alias anywhere you might use the original name:

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

Here, `AudioSample` is defined as an alias for `UInt16`. Because it's an alias, the call to `AudioSample.min` actually calls `UInt16.min`, which provides an initial value of `0` for the `maxAmplitudeFound` variable.

Booleans

Swift has a basic *Boolean* type, called `Bool`. Boolean values are referred to as *logical*, because they can only ever be true or false. Swift provides two Boolean constant values, `true` and `false`:

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

The types of `orangesAreOrange` and `turnipsAreDelicious` have been inferred as `Bool` from the fact that they were initialized with Boolean literal values. As with `Int` and `Double` above, you don't need to declare constants or variables as `Bool` if you set them to `true` or `false` as soon as you create them. Type inference helps make Swift code more concise and readable when it initializes constants or variables with other values whose type is already known.

Boolean values are particularly useful when you work with conditional statements such as the `if` statement:

```
if turnipsAreDelicious {
    print("Mmm, tasty turnips!")
} else {
    print("Eww, turnips are horrible.")
}
// Prints "Eww, turnips are horrible."
```

Conditional statements such as the `if` statement are covered in more detail in Control Flow.

Swift's type safety prevents non-Boolean values from being substituted for `Bool`. The following example reports a compile-time error:

```
let i = 1
if i {
    // this example will not compile, and will report an error
}
```

However, the alternative example below is valid:

```
let i = 1
if i == 1 {
    // this example will compile successfully
}
```

The result of the `i == 1` comparison is of type `Bool`, and so this second example passes the type-check. Comparisons like `i == 1` are discussed in Basic Operators.

As with other examples of type safety in Swift, this approach avoids accidental errors and ensures that the intention of a particular section of code is always clear.

Tuples

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and don't have to be of the same type as each other.

In this example, `(404, "Not Found")` is a tuple that describes an *HTTP status code*. An HTTP status code is a special value returned by a web

server whenever you request a web page. A status code of `404 Not Found` is returned if you request a webpage that doesn't exist.

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals
      (404, "Not Found")
```

The `(404, "Not Found")` tuple groups together an `Int` and a `String` to give the HTTP status code two separate values: a number and a human-readable description. It can be described as "a tuple of type `(Int, String)`".

You can create tuples from any permutation of types, and they can contain as many different types as you like. There's nothing stopping you from having a tuple of type `(Int, Int, Int)`, or `(String, Bool)`, or indeed any other permutation you require.

You can *decompose* a tuple's contents into separate constants or variables, which you then access as usual:

```
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// Prints "The status code is 404"
print("The status message is \(statusMessage)")
// Prints "The status message is Not Found"
```

If you only need some of the tuple's values, ignore parts of the tuple with an underscore (_) when you decompose the tuple:

```
let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")
// Prints "The status code is 404"
```

Alternatively, access the individual element values in a tuple using index numbers starting at zero:

```
print("The status code is \(http404Error.0)")
// Prints "The status code is 404"
print("The status message is \(http404Error.1)")
// Prints "The status message is Not Found"
```

You can name the individual elements in a tuple when the tuple is defined:

```
let http200Status = (statusCode: 200, description:
     "OK")
```

If you name the elements in a tuple, you can use the element names to access the values of those elements:

```
print("The status code is \
     (http200Status.statusCode)")
// Prints "The status code is 200"
print("The status message is \
     (http200Status.description)")
// Prints "The status message is OK"
```

Tuples are particularly useful as the return values of functions. A function that tries to retrieve a web page might return the `(Int, String)` tuple type

to describe the success or failure of the page retrieval. By returning a tuple with two distinct values, each of a different type, the function provides more useful information about its outcome than if it could only return a single value of a single type. For more information, see Functions with Multiple Return Values.

> NOTE
>
> Tuples are useful for simple groups of related values. They're not suited to the creation of complex data structures. If your data structure is likely to be more complex, model it as a class or structure, rather than as a tuple. For more information, see Structures and Classes.

Optionals

You use *optionals* in situations where a value may be absent. An optional represents two possibilities: Either there *is* a value, and you can unwrap the optional to access that value, or there *isn't* a value at all.

> NOTE
>
> The concept of optionals doesn't exist in C or Objective-C. The nearest thing in Objective-C is the ability to return `nil` from a method that would otherwise return an object, with `nil` meaning "the absence of a valid object." However, this only works for objects—it doesn't work for structures, basic C types, or enumeration values. For these types, Objective-C methods typically return a special value (such as `NSNotFound`) to indicate the absence of a value. This approach assumes that the method's caller knows there's a special value to test against and remembers to check for it. Swift's optionals let you indicate the absence of a value for *any type at all*, without the need for special constants.

Here's an example of how optionals can be used to cope with the absence of a value. Swift's `Int` type has an initializer which tries to convert a `String` value into an `Int` value. However, not every string can be converted into an integer. The string `"123"` can be converted into the numeric value `123`, but

the string `"hello, world"` doesn't have an obvious numeric value to convert to.

The example below uses the initializer to try to convert a `String` into an `Int`:

```
let possibleNumber = "123"
let convertedNumber = Int(possibleNumber)
// convertedNumber is inferred to be of type "Int?",
     or "optional Int"
```

Because the initializer might fail, it returns an *optional* `Int`, rather than an `Int`. An optional `Int` is written as `Int?`, not `Int`. The question mark indicates that the value it contains is optional, meaning that it might contain *some* `Int` value, or it might contain *no value at all*. (It can't contain anything else, such as a `Bool` value or a `String` value. It's either an `Int`, or it's nothing at all.)

nil

You set an optional variable to a valueless state by assigning it the special value `nil`:

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value
     of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

> NOTE
> You can't use `nil` with non-optional constants and variables. If a constant or variable in your code needs to work with the absence of a value under certain conditions, always declare it as an optional value of the appropriate type.

If you define an optional variable without providing a default value, the variable is automatically set to `nil` for you:

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

> NOTE
> Swift's `nil` isn't the same as `nil` in Objective-C. In Objective-C, `nil` is a pointer to a nonexistent object. In Swift, `nil` isn't a pointer—it's the absence of a value of a certain type. Optionals of *any* type can be set to `nil`, not just object types.

If Statements and Forced Unwrapping

You can use an `if` statement to find out whether an optional contains a value by comparing the optional against `nil`. You perform this comparison with the "equal to" operator (`==`) or the "not equal to" operator (`!=`).

If an optional has a value, it's considered to be "not equal to" `nil`:

```
if convertedNumber != nil {
    print("convertedNumber contains some integer value.")
}
// Prints "convertedNumber contains some integer value."
```

Once you're sure that the optional *does* contain a value, you can access its underlying value by adding an exclamation point (`!`) to the end of the optional's name. The exclamation point effectively says, "I know that this optional definitely has a value; please use it." This is known as *forced unwrapping* of the optional's value:

```
if convertedNumber != nil {
    print("convertedNumber has an integer value of \
     (convertedNumber!).")
}
// Prints "convertedNumber has an integer value of
     123."
```

For more about the `if` statement, see Control Flow.

> NOTE
>
> Trying to use `!` to access a nonexistent optional value triggers a runtime error. Always make sure that an optional contains a non-`nil` value before using `!` to force-unwrap its value.

Optional Binding

You use *optional binding* to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable. Optional binding can be used with `if` and `while` statements to check for a value inside an optional, and to extract that value into a constant or variable, as part of a single action. `if` and `while` statements are described in more detail in Control Flow.

Write an optional binding for an `if` statement as follows:

```
if let constantName = someOptional {
    statements
}
```

You can rewrite the `possibleNumber` example from the [Optionals](#) section to use optional binding rather than forced unwrapping:

```
if let actualNumber = Int(possibleNumber) {
    print("The string \"\(possibleNumber)\" has an integer value of \(actualNumber)")
} else {
    print("The string \"\(possibleNumber)\" couldn't be converted to an integer")
}
// Prints "The string "123" has an integer value of 123"
```

This code can be read as:

"If the optional `Int` returned by `Int(possibleNumber)` contains a value, set a new constant called `actualNumber` to the value contained in the optional."

If the conversion is successful, the `actualNumber` constant becomes available for use within the first branch of the `if` statement. It has already been initialized with the value contained *within* the optional, and so you don't use the `!` suffix to access its value. In this example, `actualNumber` is simply used to print the result of the conversion.

You can use both constants and variables with optional binding. If you wanted to manipulate the value of `actualNumber` within the first branch of

the `if` statement, you could write `if var actualNumber` instead, and the value contained within the optional would be made available as a variable rather than a constant.

You can include as many optional bindings and Boolean conditions in a single `if` statement as you need to, separated by commas. If any of the values in the optional bindings are `nil` or any Boolean condition evaluates to `false`, the whole `if` statement's condition is considered to be `false`. The following `if` statements are equivalent:

```
if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber < secondNumber && secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
// Prints "4 < 42 < 100"

if let firstNumber = Int("4") {
    if let secondNumber = Int("42") {
        if firstNumber < secondNumber && secondNumber < 100 {
            print("\(firstNumber) < \(secondNumber) < 100")
        }
    }
}
// Prints "4 < 42 < 100"
```

> NOTE
>
> Constants and variables created with optional binding in an `if` statement are available only within the body of the `if` statement. In contrast, the constants and variables created with a `guard` statement are available in the lines of code that follow the `guard` statement, as described in Early Exit.

Implicitly Unwrapped Optionals

As described above, optionals indicate that a constant or variable is allowed to have "no value". Optionals can be checked with an `if` statement to see if a value exists, and can be conditionally unwrapped with optional binding to access the optional's value if it does exist.

Sometimes it's clear from a program's structure that an optional will *always* have a value, after that value is first set. In these cases, it's useful to remove the need to check and unwrap the optional's value every time it's accessed, because it can be safely assumed to have a value all of the time.

These kinds of optionals are defined as *implicitly unwrapped optionals*. You write an implicitly unwrapped optional by placing an exclamation point (`String!`) rather than a question mark (`String?`) after the type that you want to make optional. Rather than placing an exclamation point after the optional's name when you use it, you place an exclamation point after the optional's type when you declare it.

Implicitly unwrapped optionals are useful when an optional's value is confirmed to exist immediately after the optional is first defined and can definitely be assumed to exist at every point thereafter. The primary use of implicitly unwrapped optionals in Swift is during class initialization, as described in Unowned References and Implicitly Unwrapped Optional Properties.

An implicitly unwrapped optional is a normal optional behind the scenes, but can also be used like a non-optional value, without the need to unwrap the optional value each time it's accessed. The following example shows

the difference in behavior between an optional string and an implicitly unwrapped optional string when accessing their wrapped value as an explicit `String`:

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation point

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation point
```

You can think of an implicitly unwrapped optional as giving permission for the optional to be force-unwrapped if needed. When you use an implicitly unwrapped optional value, Swift first tries to use it as an ordinary optional value; if it can't be used as an optional, Swift force-unwraps the value. In the code above, the optional value `assumedString` is force-unwrapped before assigning its value to `implicitString` because `implicitString` has an explicit, non-optional type of `String`. In code below, `optionalString` doesn't have an explicit type so it's an ordinary optional.

```
let optionalString = assumedString
// The type of optionalString is "String?" and assumedString isn't force-unwrapped.
```

If an implicitly unwrapped optional is `nil` and you try to access its wrapped value, you'll trigger a runtime error. The result is exactly the same as if you place an exclamation point after a normal optional that doesn't contain a value.

You can check whether an implicitly unwrapped optional is `nil` the same way you check a normal optional:

```
if assumedString != nil {
    print(assumedString!)
}
// Prints "An implicitly unwrapped optional string."
```

You can also use an implicitly unwrapped optional with optional binding, to check and unwrap its value in a single statement:

```
if let definiteString = assumedString {
    print(definiteString)
}
// Prints "An implicitly unwrapped optional string."
```

> NOTE
>
> Don't use an implicitly unwrapped optional when there's a possibility of a variable becoming `nil` at a later point. Always use a normal optional type if you need to check for a `nil` value during the lifetime of a variable.

Error Handling

You use *error handling* to respond to error conditions your program may encounter during execution.

In contrast to optionals, which can use the presence or absence of a value to communicate success or failure of a function, error handling allows you to

determine the underlying cause of failure, and, if necessary, propagate the error to another part of your program.

When a function encounters an error condition, it *throws* an error. That function's caller can then *catch* the error and respond appropriately.

```
func canThrowAnError() throws {
    // this function may or may not throw an error
}
```

A function indicates that it can throw an error by including the `throws` keyword in its declaration. When you call a function that can throw an error, you prepend the `try` keyword to the expression.

Swift automatically propagates errors out of their current scope until they're handled by a `catch` clause.

```
do {
    try canThrowAnError()
    // no error was thrown
} catch {
    // an error was thrown
}
```

A `do` statement creates a new containing scope, which allows errors to be propagated to one or more `catch` clauses.

Here's an example of how error handling can be used to respond to different error conditions:

```
func makeASandwich() throws {
    // ...
}

do {
    try makeASandwich()
    eatASandwich()
} catch SandwichError.outOfCleanDishes {
    washDishes()
} catch SandwichError.missingIngredients(let
     ingredients) {
    buyGroceries(ingredients)
}
```

In this example, the `makeASandwich()` function will throw an error if no clean dishes are available or if any ingredients are missing. Because `makeASandwich()` can throw an error, the function call is wrapped in a `try` expression. By wrapping the function call in a `do` statement, any errors that are thrown will be propagated to the provided `catch` clauses.

If no error is thrown, the `eatASandwich()` function is called. If an error is thrown and it matches the `SandwichError.outOfCleanDishes` case, then the `washDishes()` function will be called. If an error is thrown and it matches the `SandwichError.missingIngredients` case, then the `buyGroceries(_:)` function is called with the associated `[String]` value captured by the `catch` pattern.

Throwing, catching, and propagating errors is covered in greater detail in [Error Handling](#).

Assertions and Preconditions

Assertions and *preconditions* are checks that happen at runtime. You use them to make sure an essential condition is satisfied before executing any further code. If the Boolean condition in the assertion or precondition evaluates to `true`, code execution continues as usual. If the condition evaluates to `false`, the current state of the program is invalid; code execution ends, and your app is terminated.

You use assertions and preconditions to express the assumptions you make and the expectations you have while coding, so you can include them as part of your code. Assertions help you find mistakes and incorrect assumptions during development, and preconditions help you detect issues in production.

In addition to verifying your expectations at runtime, assertions and preconditions also become a useful form of documentation within the code. Unlike the error conditions discussed in Error Handling above, assertions and preconditions aren't used for recoverable or expected errors. Because a failed assertion or precondition indicates an invalid program state, there's no way to catch a failed assertion.

Using assertions and preconditions isn't a substitute for designing your code in such a way that invalid conditions are unlikely to arise. However, using them to enforce valid data and state causes your app to terminate more predictably if an invalid state occurs, and helps make the problem easier to debug. Stopping execution as soon as an invalid state is detected also helps limit the damage caused by that invalid state.

The difference between assertions and preconditions is in when they're checked: Assertions are checked only in debug builds, but preconditions are checked in both debug and production builds. In production builds, the condition inside an assertion isn't evaluated. This means you can use as many assertions as you want during your development process, without impacting performance in production.

Debugging with Assertions

You write an assertion by calling the `assert(_:_:file:line:)` function from the Swift standard library. You pass this function an expression that evaluates to `true` or `false` and a message to display if the result of the condition is `false`. For example:

```
let age = -3
assert(age >= 0, "A person's age can't be less than zero.")
// This assertion fails because -3 isn't >= 0.
```

In this example, code execution continues if `age >= 0` evaluates to `true`, that is, if the value of `age` is nonnegative. If the value of `age` is negative, as in the code above, then `age >= 0` evaluates to `false`, and the assertion fails, terminating the application.

You can omit the assertion message—for example, when it would just repeat the condition as prose.

```
assert(age >= 0)
```

If the code already checks the condition, you use the `assertionFailure(_:file:line:)` function to indicate that an assertion has failed. For example:

```
if age > 10 {
    print("You can ride the roller-coaster or the ferris wheel.")
} else if age >= 0 {
    print("You can ride the ferris wheel.")
} else {
    assertionFailure("A person's age can't be less than zero.")
}
```

Enforcing Preconditions

Use a precondition whenever a condition has the potential to be false, but must *definitely* be true for your code to continue execution. For example, use a precondition to check that a subscript isn't out of bounds, or to check that a function has been passed a valid value.

You write a precondition by calling the `precondition(_:_:file:line:)` function. You pass this function an expression that evaluates to `true` or `false` and a message to display if the result of the condition is `false`. For example:

```
// In the implementation of a subscript...
precondition(index > 0, "Index must be greater than zero.")
```

You can also call the `preconditionFailure(_:file:line:)` function to indicate that a failure has occurred—for example, if the default case of a switch was taken, but all valid input data should have been handled by one of the switch's other cases.

> NOTE
>
> If you compile in unchecked mode (`-Ounchecked`), preconditions aren't checked. The compiler assumes that preconditions are always true, and it optimizes your code accordingly. However, the `fatalError(_:file:line:)` function always halts execution, regardless of optimization settings.
>
> You can use the `fatalError(_:file:line:)` function during prototyping and early development to create stubs for functionality that hasn't been implemented yet, by writing `fatalError("Unimplemented")` as the stub implementation. Because fatal errors are never optimized out, unlike assertions or preconditions, you can be sure that execution always halts if it encounters a stub implementation.

Converted by Evan at Apps Dissected -www.appsdissected.com

Basic Operators

An *operator* is a special symbol or phrase that you use to check, change, or combine values. For example, the addition operator (`+`) adds two numbers, as in `let i = 1 + 2`, and the logical AND operator (`&&`) combines two Boolean values, as in `if enteredDoorCode && passedRetinaScan`.

Swift supports the operators you may already know from languages like C, and improves several capabilities to eliminate common coding errors. The assignment operator (`=`) doesn't return a value, to prevent it from being mistakenly used when the equal to operator (`==`) is intended. Arithmetic operators (`+`, `-`, `*`, `/`, `%` and so forth) detect and disallow value overflow, to avoid unexpected results when working with numbers that become larger or smaller than the allowed value range of the type that stores them. You can opt in to value overflow behavior by using Swift's overflow operators, as described in Overflow Operators.

Swift also provides range operators that aren't found in C, such as `a..<b` and `a...b`, as a shortcut for expressing a range of values.

This chapter describes the common operators in Swift. Advanced Operators covers Swift's advanced operators, and describes how to define your own custom operators and implement the standard operators for your own custom types.

Terminology

Operators are unary, binary, or ternary:

- *Unary* operators operate on a single target (such as `-a`). Unary *prefix* operators appear immediately before their target (such as `!b`), and

unary *postfix* operators appear immediately after their target (such as `c!`).

- *Binary* operators operate on two targets (such as `2 + 3`) and are *infix* because they appear in between their two targets.
- *Ternary* operators operate on three targets. Like C, Swift has only one ternary operator, the ternary conditional operator (`a ? b : c`).

The values that operators affect are *operands*. In the expression `1 + 2`, the `+` symbol is a binary operator and its two operands are the values `1` and `2`.

Assignment Operator

The *assignment operator* (`a = b`) initializes or updates the value of `a` with the value of `b`:

```
let b = 10
var a = 5
a = b
// a is now equal to 10
```

If the right side of the assignment is a tuple with multiple values, its elements can be decomposed into multiple constants or variables at once:

```
let (x, y) = (1, 2)
// x is equal to 1, and y is equal to 2
```

Unlike the assignment operator in C and Objective-C, the assignment operator in Swift doesn't itself return a value. The following statement isn't valid:

```
if x = y {
    // This isn't valid, because x = y doesn't return a value.
}
```

This feature prevents the assignment operator (`=`) from being used by accident when the equal to operator (`==`) is actually intended. By making `if x = y` invalid, Swift helps you to avoid these kinds of errors in your code.

Arithmetic Operators

Swift supports the four standard *arithmetic operators* for all number types:

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Division (`/`)

```
1 + 2       // equals 3
5 - 3       // equals 2
2 * 3       // equals 6
10.0 / 2.5  // equals 4.0
```

Unlike the arithmetic operators in C and Objective-C, the Swift arithmetic operators don't allow values to overflow by default. You can opt in to value overflow behavior by using Swift's overflow operators (such as `a &+ b`). See Overflow Operators.

The addition operator is also supported for `String` concatenation:

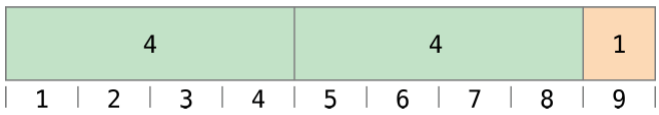
```
"hello, " + "world"  // equals "hello, world"
```

Remainder Operator

The *remainder operator* (`a % b`) works out how many multiples of `b` will fit inside `a` and returns the value that's left over (known as the *remainder*).

> NOTE
>
> The remainder operator (`%`) is also known as a *modulo operator* in other languages. However, its behavior in Swift for negative numbers means that, strictly speaking, it's a remainder rather than a modulo operation.

Here's how the remainder operator works. To calculate `9 % 4`, you first work out how many `4`s will fit inside `9`:



You can fit two `4`s inside `9`, and the remainder is `1` (shown in orange).

In Swift, this would be written as:

```
9 % 4    // equals 1
```

To determine the answer for `a % b`, the `%` operator calculates the following equation and returns `remainder` as its output:

`a` = (`b` x `some multiplier`) + `remainder`

where `some multiplier` is the largest number of multiples of `b` that will fit inside `a`.

Inserting 9 and 4 into this equation yields:

9 = (4 x 2) + 1

The same method is applied when calculating the remainder for a negative value of a:

```
-9 % 4   // equals -1
```

Inserting −9 and 4 into the equation yields:

−9 = (4 x −2) + −1

giving a remainder value of −1.

The sign of b is ignored for negative values of b. This means that a % b and a % −b always give the same answer.

Unary Minus Operator

The sign of a numeric value can be toggled using a prefixed −, known as the *unary minus operator*:

```
let three = 3
let minusThree = -three       // minusThree equals -3
let plusThree = -minusThree   // plusThree equals 3, or "minus minus three"
```

The unary minus operator (−) is prepended directly before the value it operates on, without any white space.

Unary Plus Operator
The *unary plus operator* (+) simply returns the value it operates on, without any change:

```
let minusSix = -6
let alsoMinusSix = +minusSix  // alsoMinusSix equals -6
```

Although the unary plus operator doesn't actually do anything, you can use it to provide symmetry in your code for positive numbers when also using the unary minus operator for negative numbers.

Compound Assignment Operators

Like C, Swift provides *compound assignment operators* that combine assignment (=) with another operation. One example is the *addition assignment operator* (+=):

```
var a = 1
a += 2
// a is now equal to 3
```

The expression `a += 2` is shorthand for `a = a + 2`. Effectively, the addition and the assignment are combined into one operator that performs both tasks at the same time.

> NOTE
> The compound assignment operators don't return a value. For example, you can't write `let b = a += 2`.

For information about the operators provided by the Swift standard library, see Operator Declarations.

Comparison Operators

Swift supports the following comparison operators:

- Equal to (`a == b`)
- Not equal to (`a != b`)
- Greater than (`a > b`)
- Less than (`a < b`)
- Greater than or equal to (`a >= b`)
- Less than or equal to (`a <= b`)

> NOTE
>
> Swift also provides two *identity operators* (`===` and `!==`), which you use to test whether two object references both refer to the same object instance. For more information, see Identity Operators.

Each of the comparison operators returns a `Bool` value to indicate whether or not the statement is true:

```
1 == 1   // true because 1 is equal to 1
2 != 1   // true because 2 isn't equal to 1
2 > 1    // true because 2 is greater than 1
1 < 2    // true because 1 is less than 2
1 >= 1   // true because 1 is greater than or equal to 1
2 <= 1   // false because 2 isn't less than or equal to 1
```

Comparison operators are often used in conditional statements, such as the `if` statement:

```
let name = "world"
if name == "world" {
    print("hello, world")
} else {
    print("I'm sorry \(name), but I don't recognize you")
}
// Prints "hello, world", because name is indeed equal to "world".
```

For more about the `if` statement, see Control Flow.

You can compare two tuples if they have the same type and the same number of values. Tuples are compared from left to right, one value at a time, until the comparison finds two values that aren't equal. Those two values are compared, and the result of that comparison determines the

overall result of the tuple comparison. If all the elements are equal, then the tuples themselves are equal. For example:

```
(1, "zebra") < (2, "apple")   // true because 1 is less than 2; "zebra" and "apple" aren't compared
(3, "apple") < (3, "bird")    // true because 3 is equal to 3, and "apple" is less than "bird"
(4, "dog") == (4, "dog")      // true because 4 is equal to 4, and "dog" is equal to "dog"
```

In the example above, you can see the left-to-right comparison behavior on the first line. Because `1` is less than `2`, `(1, "zebra")` is considered less than `(2, "apple")`, regardless of any other values in the tuples. It doesn't matter that `"zebra"` isn't less than `"apple"`, because the comparison is already determined by the tuples' first elements. However, when the tuples' first elements are the same, their second elements *are* compared—this is what happens on the second and third line.

Tuples can be compared with a given operator only if the operator can be applied to each value in the respective tuples. For example, as demonstrated in the code below, you can compare two tuples of type `(String, Int)` because both `String` and `Int` values can be compared using the `<` operator. In contrast, two tuples of type `(String, Bool)` can't be compared with the `<` operator because the `<` operator can't be applied to `Bool` values.

```
("blue", -1) < ("purple", 1)        // OK, evaluates to true
("blue", false) < ("purple", true)  // Error because < can't compare Boolean values
```

> NOTE
>
> The Swift standard library includes tuple comparison operators for tuples with fewer than seven elements. To compare tuples with seven or more elements, you must implement the comparison operators yourself.

Ternary Conditional Operator

The *ternary conditional operator* is a special operator with three parts, which takes the form `question ? answer1 : answer2`. It's a shortcut for evaluating one of two expressions based on whether `question` is true or false. If `question` is true, it evaluates `answer1` and returns its value; otherwise, it evaluates `answer2` and returns its value.

The ternary conditional operator is shorthand for the code below:

```
if question {
    answer1
} else {
    answer2
}
```

Here's an example, which calculates the height for a table row. The row height should be 50 points taller than the content height if the row has a header, and 20 points taller if the row doesn't have a header:

```
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 :
     20)
// rowHeight is equal to 90
```

The example above is shorthand for the code below:

```
let contentHeight = 40
let hasHeader = true
let rowHeight: Int
if hasHeader {
    rowHeight = contentHeight + 50
} else {
    rowHeight = contentHeight + 20
}
// rowHeight is equal to 90
```

The first example's use of the ternary conditional operator means that `rowHeight` can be set to the correct value on a single line of code, which is more concise than the code used in the second example.

The ternary conditional operator provides an efficient shorthand for deciding which of two expressions to consider. Use the ternary conditional operator with care, however. Its conciseness can lead to hard-to-read code if overused. Avoid combining multiple instances of the ternary conditional operator into one compound statement.

Nil-Coalescing Operator

The *nil-coalescing operator* (`a ?? b`) unwraps an optional `a` if it contains a value, or returns a default value `b` if `a` is `nil`. The expression `a` is always of an optional type. The expression `b` must match the type that's stored inside `a`.

The nil-coalescing operator is shorthand for the code below:

```
a != nil ? a! : b
```

The code above uses the ternary conditional operator and forced unwrapping (`a!`) to access the value wrapped inside `a` when `a` isn't `nil`, and to return `b` otherwise. The nil-coalescing operator provides a more elegant way to encapsulate this conditional checking and unwrapping in a concise and readable form.

> NOTE
>
> If the value of `a` is non-`nil`, the value of `b` isn't evaluated. This is known as *short-circuit evaluation*.

The example below uses the nil-coalescing operator to choose between a default color name and an optional user-defined color name:

```
let defaultColorName = "red"
var userDefinedColorName: String?   // defaults to nil

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is nil, so colorNameToUse is set to the default of "red"
```

The `userDefinedColorName` variable is defined as an optional `String`, with a default value of `nil`. Because `userDefinedColorName` is of an optional type, you can use the nil-coalescing operator to consider its value. In the example above, the operator is used to determine an initial value for a `String` variable called `colorNameToUse`. Because `userDefinedColorName` is `nil`, the expression `userDefinedColorName ?? defaultColorName` returns the value of `defaultColorName`, or `"red"`.

If you assign a non-`nil` value to `userDefinedColorName` and perform the nil-coalescing operator check again, the value wrapped inside `userDefinedColorName` is used instead of the default:

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName isn't nil, so colorNameToUse is set to "green"
```

Range Operators

Swift includes several *range operators*, which are shortcuts for expressing a range of values.

Closed Range Operator

The *closed range operator* (`a...b`) defines a range that runs from `a` to `b`, and includes the values `a` and `b`. The value of `a` must not be greater than `b`.

The closed range operator is useful when iterating over a range in which you want all of the values to be used, such as with a `for-in` loop:

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

For more about `for-in` loops, see Control Flow.

Half-Open Range Operator

The *half-open range operator* (`a..<b`) defines a range that runs from `a` to `b`, but doesn't include `b`. It's said to be *half-open* because it contains its first value, but not its final value. As with the closed range operator, the value of `a` must not be greater than `b`. If the value of `a` is equal to `b`, then the resulting range will be empty.

Half-open ranges are particularly useful when you work with zero-based lists such as arrays, where it's useful to count up to (but not including) the

length of the list:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count {
    print("Person \(i + 1) is called \(names[i])")
}
// Person 1 is called Anna
// Person 2 is called Alex
// Person 3 is called Brian
// Person 4 is called Jack
```

Note that the array contains four items, but `0..<count` only counts as far as `3` (the index of the last item in the array), because it's a half-open range. For more about arrays, see Arrays.

One-Sided Ranges

The closed range operator has an alternative form for ranges that continue as far as possible in one direction—for example, a range that includes all the elements of an array from index 2 to the end of the array. In these cases, you can omit the value from one side of the range operator. This kind of range is called a *one-sided range* because the operator has a value on only one side. For example:

```
for name in names[2...] {
    print(name)
}
// Brian
// Jack

for name in names[...2] {
    print(name)
}
// Anna
// Alex
// Brian
```

The half-open range operator also has a one-sided form that's written with only its final value. Just like when you include a value on both sides, the final value isn't part of the range. For example:

```
for name in names[..<2] {
    print(name)
}
// Anna
// Alex
```

One-sided ranges can be used in other contexts, not just in subscripts. You can't iterate over a one-sided range that omits a first value, because it isn't clear where iteration should begin. You *can* iterate over a one-sided range that omits its final value; however, because the range continues indefinitely, make sure you add an explicit end condition for the loop. You can also

check whether a one-sided range contains a particular value, as shown in the code below.

```
let range = ...5
range.contains(7)   // false
range.contains(4)   // true
range.contains(-1)  // true
```

Logical Operators

Logical operators modify or combine the Boolean logic values `true` and `false`. Swift supports the three standard logical operators found in C-based languages:

- Logical NOT (`!a`)
- Logical AND (`a && b`)
- Logical OR (`a || b`)

Logical NOT Operator

The *logical NOT operator* (`!a`) inverts a Boolean value so that `true` becomes `false`, and `false` becomes `true`.

The logical NOT operator is a prefix operator, and appears immediately before the value it operates on, without any white space. It can be read as "not `a`", as seen in the following example:

```
let allowedEntry = false
if !allowedEntry {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

The phrase `if !allowedEntry` can be read as "if not allowed entry." The subsequent line is only executed if "not allowed entry" is true; that is, if `allowedEntry` is `false`.

As in this example, careful choice of Boolean constant and variable names can help to keep code readable and concise, while avoiding double negatives or confusing logic statements.

Logical AND Operator

The *logical AND operator* (`a && b`) creates logical expressions where both values must be `true` for the overall expression to also be `true`.

If either value is `false`, the overall expression will also be `false`. In fact, if the *first* value is `false`, the second value won't even be evaluated, because it can't possibly make the overall expression equate to `true`. This is known as *short-circuit evaluation*.

This example considers two `Bool` values and only allows access if both values are `true`:

```swift
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

Logical OR Operator

The *logical OR operator* (`a || b`) is an infix operator made from two adjacent pipe characters. You use it to create logical expressions in which only *one* of the two values has to be `true` for the overall expression to be `true`.

Like the Logical AND operator above, the Logical OR operator uses short-circuit evaluation to consider its expressions. If the left side of a Logical OR expression is `true`, the right side isn't evaluated, because it can't change the outcome of the overall expression.

In the example below, the first `Bool` value (`hasDoorKey`) is `false`, but the second value (`knowsOverridePassword`) is `true`. Because one value is `true`, the overall expression also evaluates to `true`, and access is allowed:

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

Combining Logical Operators
You can combine multiple logical operators to create longer compound expressions:

```
if enteredDoorCode && passedRetinaScan || hasDoorKey
     || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

This example uses multiple `&&` and `||` operators to create a longer compound expression. However, the `&&` and `||` operators still operate on only two values, so this is actually three smaller expressions chained together. The example can be read as:

If we've entered the correct door code and passed the retina scan, or if we have a valid door key, or if we know the emergency override password,

then allow access.

Based on the values of `enteredDoorCode`, `passedRetinaScan`, and `hasDoorKey`, the first two subexpressions are `false`. However, the emergency override password is known, so the overall compound expression still evaluates to `true`.

> NOTE
>
> The Swift logical operators `&&` and `||` are left-associative, meaning that compound expressions with multiple logical operators evaluate the leftmost subexpression first.

Explicit Parentheses

It's sometimes useful to include parentheses when they're not strictly needed, to make the intention of a complex expression easier to read. In the door access example above, it's useful to add parentheses around the first part of the compound expression to make its intent explicit:

```
if (enteredDoorCode && passedRetinaScan) ||
     hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

The parentheses make it clear that the first two values are considered as part of a separate possible state in the overall logic. The output of the compound expression doesn't change, but the overall intention is clearer to the reader. Readability is always preferred over brevity; use parentheses where they help to make your intentions clear.

Strings and Characters

A *string* is a series of characters, such as `"hello, world"` or `"albatross"`. Swift strings are represented by the `String` type. The contents of a `String` can be accessed in various ways, including as a collection of `Character` values.

Swift's `String` and `Character` types provide a fast, Unicode-compliant way to work with text in your code. The syntax for string creation and manipulation is lightweight and readable, with a string literal syntax that's similar to C. String concatenation is as simple as combining two strings with the `+` operator, and string mutability is managed by choosing between a constant or a variable, just like any other value in Swift. You can also use strings to insert constants, variables, literals, and expressions into longer strings, in a process known as string interpolation. This makes it easy to create custom string values for display, storage, and printing.

Despite this simplicity of syntax, Swift's `String` type is a fast, modern string implementation. Every string is composed of encoding-independent Unicode characters, and provides support for accessing those characters in various Unicode representations.

> NOTE
>
> Swift's `String` type is bridged with Foundation's `NSString` class. Foundation also extends `String` to expose methods defined by `NSString`. This means, if you import Foundation, you can access those `NSString` methods on `String` without casting.
>
> For more information about using `String` with Foundation and Cocoa, see Bridging Between String and NSString.

String Literals

You can include predefined `String` values within your code as *string literals*. A string literal is a sequence of characters surrounded by double quotation marks (`"`).

Use a string literal as an initial value for a constant or variable:

```
let someString = "Some string literal value"
```

Note that Swift infers a type of `String` for the `someString` constant because it's initialized with a string literal value.

Multiline String Literals

If you need a string that spans several lines, use a multiline string literal—a sequence of characters surrounded by three double quotation marks:

```
let quotation = """
The White Rabbit put on his spectacles.  "Where
     shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely,
     "and go on
till you come to the end; then stop."
"""
```

A multiline string literal includes all of the lines between its opening and closing quotation marks. The string begins on the first line after the opening quotation marks (`"""`) and ends on the line before the closing quotation marks, which means that neither of the strings below start or end with a line break:

```
let singleLineString = "These are the same."
let multilineString = """
These are the same.
"""
```

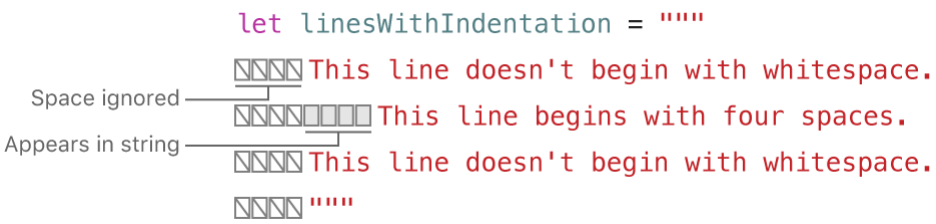
When your source code includes a line break inside of a multiline string literal, that line break also appears in the string's value. If you want to use line breaks to make your source code easier to read, but you don't want the line breaks to be part of the string's value, write a backslash (\) at the end of those lines:

```
let softWrappedQuotation = """
The White Rabbit put on his spectacles.  "Where
     shall I begin, \
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely,
     "and go on \
till you come to the end; then stop."
"""
```

To make a multiline string literal that begins or ends with a line feed, write a blank line as the first or last line. For example:

```
let lineBreaks = """

This string starts with a line break.
It also ends with a line break.

"""
```

A multiline string can be indented to match the surrounding code. The whitespace before the closing quotation marks (`"""`) tells Swift what whitespace to ignore before all of the other lines. However, if you write whitespace at the beginning of a line in addition to what's before the closing quotation marks, that whitespace *is* included.



In the example above, even though the entire multiline string literal is indented, the first and last lines in the string don't begin with any whitespace. The middle line has more indentation than the closing quotation marks, so it starts with that extra four-space indentation.

Special Characters in String Literals

String literals can include the following special characters:

- The escaped special characters `\0` (null character), `\\` (backslash), `\t` (horizontal tab), `\n` (line feed), `\r` (carriage return), `\"` (double quotation mark) and `\'` (single quotation mark)

- An arbitrary Unicode scalar value, written as `\u{n}`, where *n* is a 1–8 digit hexadecimal number (Unicode is discussed in Unicode below)

The code below shows four examples of these special characters. The `wiseWords` constant contains two escaped double quotation marks. The `dollarSign`, `blackHeart`, and `sparklingHeart` constants demonstrate the Unicode scalar format:

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein
let dollarSign = "\u{24}"        // $,  Unicode scalar U+0024
let blackHeart = "\u{2665}"      // ♥,  Unicode scalar U+2665
let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496
```

Because multiline string literals use three double quotation marks instead of just one, you can include a double quotation mark (`"`) inside of a multiline string literal without escaping it. To include the text `"""` in a multiline string, escape at least one of the quotation marks. For example:

```
let threeDoubleQuotationMarks = """
Escaping the first quotation mark \"""
Escaping all three quotation marks \"\"\"
"""
```

Extended String Delimiters
You can place a string literal within *extended delimiters* to include special characters in a string without invoking their effect. You place your string within quotation marks (`"`) and surround that with number signs (`#`). For example, printing the string literal `#"Line 1\nLine 2"#` prints the line feed escape sequence (`\n`) rather than printing the string across two lines.

If you need the special effects of a character in a string literal, match the number of number signs within the string following the escape character (`\`). For example, if your string is `#"Line 1\nLine 2"#` and you want to break the line, you can use `#"Line 1\#nLine 2"#` instead. Similarly, `###"Line1\###nLine2"###` also breaks the line.

String literals created using extended delimiters can also be multiline string literals. You can use extended delimiters to include the text `"""` in a multiline string, overriding the default behavior that ends the literal. For example:

```
let threeMoreDoubleQuotationMarks = #"""
Here are three more double quotes: """
"""#
```

Initializing an Empty String

To create an empty `String` value as the starting point for building a longer string, either assign an empty string literal to a variable, or initialize a new `String` instance with initializer syntax:

```
var emptyString = ""                // empty string literal
var anotherEmptyString = String()  // initializer syntax
// these two strings are both empty, and are equivalent to each other
```

Find out whether a `String` value is empty by checking its Boolean `isEmpty` property:

```
if emptyString.isEmpty {
    print("Nothing to see here")
}
// Prints "Nothing to see here"
```

String Mutability

You indicate whether a particular `String` can be modified (or *mutated*) by assigning it to a variable (in which case it can be modified), or to a constant (in which case it can't be modified):

```
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"

let constantString = "Highlander"
constantString += " and another Highlander"
// this reports a compile-time error - a constant
     string cannot be modified
```

> NOTE
>
> This approach is different from string mutation in Objective-C and Cocoa, where you choose between two classes (`NSString` and `NSMutableString`) to indicate whether a string can be mutated.

Strings Are Value Types

Swift's `String` type is a *value type*. If you create a new `String` value, that `String` value is *copied* when it's passed to a function or method, or when it's assigned to a constant or variable. In each case, a new copy of the existing `String` value is created, and the new copy is passed or assigned, not the original version. Value types are described in Structures and Enumerations Are Value Types.

Swift's copy-by-default `String` behavior ensures that when a function or method passes you a `String` value, it's clear that you own that exact `String` value, regardless of where it came from. You can be confident that the string you are passed won't be modified unless you modify it yourself.

Behind the scenes, Swift's compiler optimizes string usage so that actual copying takes place only when absolutely necessary. This means you always get great performance when working with strings as value types.

Working with Characters

You can access the individual `Character` values for a `String` by iterating over the string with a `for-in` loop:

```
for character in "Dog!🐶" {
    print(character)
}
// D
// o
// g
// !
// 🐶
```

The `for-in` loop is described in [For-In Loops](#).

Alternatively, you can create a stand-alone `Character` constant or variable from a single-character string literal by providing a `Character` type annotation:

```
let exclamationMark: Character = "!"
```

`String` values can be constructed by passing an array of `Character` values as an argument to its initializer:

```
let catCharacters: [Character] = ["C", "a", "t",
     "!", "🐱"]
let catString = String(catCharacters)
print(catString)
// Prints "Cat!🐱"
```

Concatenating Strings and Characters

`String` values can be added together (or *concatenated*) with the addition operator (`+`) to create a new `String` value:

```
let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2
// welcome now equals "hello there"
```

You can also append a `String` value to an existing `String` variable with the addition assignment operator (`+=`):

```
var instruction = "look over"
instruction += string2
// instruction now equals "look over there"
```

You can append a `Character` value to a `String` variable with the `String` type's `append()` method:

```
let exclamationMark: Character = "!"
welcome.append(exclamationMark)
// welcome now equals "hello there!"
```

> NOTE
>
> You can't append a `String` or `Character` to an existing `Character` variable, because a `Character` value must contain a single character only.

If you're using multiline string literals to build up the lines of a longer string, you want every line in the string to end with a line break, including the last line. For example:

```
let badStart = """
one
two
"""
let end = """
three
"""
print(badStart + end)
// Prints two lines:
// one
// twothree

let goodStart = """
one
two

"""
print(goodStart + end)
// Prints three lines:
// one
// two
// three
```

In the code above, concatenating `badStart` with `end` produces a two-line string, which isn't the desired result. Because the last line of `badStart` doesn't end with a line break, that line gets combined with the first line of

`end`. In contrast, both lines of `goodStart` end with a line break, so when it's combined with `end` the result has three lines, as expected.

String Interpolation

String interpolation is a way to construct a new `String` value from a mix of constants, variables, literals, and expressions by including their values inside a string literal. You can use string interpolation in both single-line and multiline string literals. Each item that you insert into the string literal is wrapped in a pair of parentheses, prefixed by a backslash (`\`):

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \
     (Double(multiplier) * 2.5)"
// message is "3 times 2.5 is 7.5"
```

In the example above, the value of `multiplier` is inserted into a string literal as `\(multiplier)`. This placeholder is replaced with the actual value of `multiplier` when the string interpolation is evaluated to create an actual string.

The value of `multiplier` is also part of a larger expression later in the string. This expression calculates the value of `Double(multiplier) * 2.5` and inserts the result (`7.5`) into the string. In this case, the expression is written as `\(Double(multiplier) * 2.5)` when it's included inside the string literal.

You can use extended string delimiters to create strings containing characters that would otherwise be treated as a string interpolation. For example:

```
print(#"Write an interpolated string in Swift using \(multiplier)."#)
// Prints "Write an interpolated string in Swift using \(multiplier)."
```

To use string interpolation inside a string that uses extended delimiters, match the number of number signs after the backslash to the number of number signs at the beginning and end of the string. For example:

```
print(#"6 times 7 is \#(6 * 7)."#)
// Prints "6 times 7 is 42."
```

> NOTE
>
> The expressions you write inside parentheses within an interpolated string can't contain an unescaped backslash (\), a carriage return, or a line feed. However, they can contain other string literals.

Unicode

Unicode is an international standard for encoding, representing, and processing text in different writing systems. It enables you to represent almost any character from any language in a standardized form, and to read and write those characters to and from an external source such as a text file or web page. Swift's `String` and `Character` types are fully Unicode-compliant, as described in this section.

Unicode Scalar Values

Behind the scenes, Swift's native `String` type is built from *Unicode scalar values*. A Unicode scalar value is a unique 21-bit number for a character or modifier, such as `U+0061` for `LATIN SMALL LETTER A` (`"a"`), or `U+1F425` for `FRONT-FACING BABY CHICK` (`"🐥"`).

Note that not all 21-bit Unicode scalar values are assigned to a character—some scalars are reserved for future assignment or for use in UTF-16 encoding. Scalar values that have been assigned to a character typically also have a name, such as `LATIN SMALL LETTER A` and `FRONT-FACING BABY CHICK` in the examples above.

Extended Grapheme Clusters

Every instance of Swift's `Character` type represents a single *extended grapheme cluster*. An extended grapheme cluster is a sequence of one or more Unicode scalars that (when combined) produce a single human-readable character.

Here's an example. The letter `é` can be represented as the single Unicode scalar `é` (`LATIN SMALL LETTER E WITH ACUTE`, or `U+00E9`). However, the same letter can also be represented as a *pair* of scalars—a standard letter `e` (`LATIN SMALL LETTER E`, or `U+0065`), followed by the `COMBINING ACUTE ACCENT` scalar (`U+0301`). The `COMBINING ACUTE ACCENT` scalar is graphically applied to the scalar that precedes it, turning an `e` into an `é` when it's rendered by a Unicode-aware text-rendering system.

In both cases, the letter `é` is represented as a single Swift `Character` value that represents an extended grapheme cluster. In the first case, the cluster contains a single scalar; in the second case, it's a cluster of two scalars:

```swift
let eAcute: Character = "\u{E9}"
    // é
let combinedEAcute: Character = "\u{65}\u{301}"
    // e followed by ́
// eAcute is é, combinedEAcute is é
```

Extended grapheme clusters are a flexible way to represent many complex script characters as a single `Character` value. For example, Hangul syllables from the Korean alphabet can be represented as either a precomposed or decomposed sequence. Both of these representations qualify as a single `Character` value in Swift:

```swift
let precomposed: Character = "\u{D55C}"
    // 한
let decomposed: Character =
    "\u{1112}\u{1161}\u{11AB}"   // ㅎ, ㅏ, ㄴ
// precomposed is 한, decomposed is 한
```

Extended grapheme clusters enable scalars for enclosing marks (such as `COMBINING ENCLOSING CIRCLE`, or `U+20DD`) to enclose other Unicode scalars as part of a single `Character` value:

```swift
let enclosedEAcute: Character = "\u{E9}\u{20DD}"
// enclosedEAcute is é⃝
```

Unicode scalars for regional indicator symbols can be combined in pairs to make a single `Character` value, such as this combination of `REGIONAL INDICATOR SYMBOL LETTER U` (`U+1F1FA`) and `REGIONAL INDICATOR SYMBOL LETTER S` (`U+1F1F8`):

```
let regionalIndicatorForUS: Character =
     "\u{1F1FA}\u{1F1F8}"
// regionalIndicatorForUS is 🇺🇸
```

Counting Characters

To retrieve a count of the `Character` values in a string, use the `count` property of the string:

```
let unusualMenagerie = "Koala 🨨, Snail 🐌, Penguin
     🐧, Dromedary 🐪"
print("unusualMenagerie has \
     (unusualMenagerie.count) characters")
// Prints "unusualMenagerie has 40 characters"
```

Note that Swift's use of extended grapheme clusters for `Character` values means that string concatenation and modification may not always affect a string's character count.

For example, if you initialize a new string with the four-character word `cafe`, and then append a `COMBINING ACUTE ACCENT` (`U+0301`) to the end of the string, the resulting string will still have a character count of `4`, with a fourth character of `é`, not `e`:

```
var word = "cafe"
print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in cafe is 4"

word += "\u{301}"    // COMBINING ACUTE ACCENT, U+0301

print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in café is 4"
```

> NOTE
>
> Extended grapheme clusters can be composed of multiple Unicode scalars. This means that different characters—and different representations of the same character—can require different amounts of memory to store. Because of this, characters in Swift don't each take up the same amount of memory within a string's representation. As a result, the number of characters in a string can't be calculated without iterating through the string to determine its extended grapheme cluster boundaries. If you are working with particularly long string values, be aware that the `count` property must iterate over the Unicode scalars in the entire string in order to determine the characters for that string.
>
> The count of the characters returned by the `count` property isn't always the same as the `length` property of an `NSString` that contains the same characters. The length of an `NSString` is based on the number of 16-bit code units within the string's UTF-16 representation and not the number of Unicode extended grapheme clusters within the string.

Accessing and Modifying a String

You access and modify a string through its methods and properties, or by using subscript syntax.

String Indices

Each `String` value has an associated *index type*, `String.Index`, which corresponds to the position of each `Character` in the string.

As mentioned above, different characters can require different amounts of memory to store, so in order to determine which `Character` is at a particular position, you must iterate over each Unicode scalar from the start or end of that `String`. For this reason, Swift strings can't be indexed by integer values.

Use the `startIndex` property to access the position of the first `Character` of a `String`. The `endIndex` property is the position after the last character in a `String`. As a result, the `endIndex` property isn't a valid argument to a string's subscript. If a `String` is empty, `startIndex` and `endIndex` are equal.

You access the indices before and after a given index using the `index(before:)` and `index(after:)` methods of `String`. To access an index farther away from the given index, you can use the `index(_:offsetBy:)` method instead of calling one of these methods multiple times.

You can use subscript syntax to access the `Character` at a particular `String` index.

```
let greeting = "Guten Tag!"
greeting[greeting.startIndex]
// G
greeting[greeting.index(before: greeting.endIndex)]
// !
greeting[greeting.index(after: greeting.startIndex)]
// u
let index = greeting.index(greeting.startIndex,
     offsetBy: 7)
greeting[index]
// a
```

Attempting to access an index outside of a string's range or a `Character` at an index outside of a string's range will trigger a runtime error.

```
greeting[greeting.endIndex] // Error
greeting.index(after: greeting.endIndex) // Error
```

Use the `indices` property to access all of the indices of individual characters in a string.

```
for index in greeting.indices {
    print("\(greeting[index]) ", terminator: "")
}
// Prints "G u t e n   T a g ! "
```

> NOTE
>
> You can use the `startIndex` and `endIndex` properties and the `index(before:)`, `index(after:)`, and `index(_:offsetBy:)` methods on any type that conforms to the `Collection` protocol. This includes `String`, as shown here, as well as collection types such as `Array`, `Dictionary`, and `Set`.

Inserting and Removing

To insert a single character into a string at a specified index, use the `insert(_:at:)` method, and to insert the contents of another string at a specified index, use the `insert(contentsOf:at:)` method.

```
var welcome = "hello"
welcome.insert("!", at: welcome.endIndex)
// welcome now equals "hello!"

welcome.insert(contentsOf: " there", at:
     welcome.index(before: welcome.endIndex))
// welcome now equals "hello there!"
```

To remove a single character from a string at a specified index, use the `remove(at:)` method, and to remove a substring at a specified range, use the `removeSubrange(_:)` method:

```
welcome.remove(at: welcome.index(before:
     welcome.endIndex))
// welcome now equals "hello there"

let range = welcome.index(welcome.endIndex,
     offsetBy: -6)..<welcome.endIndex
welcome.removeSubrange(range)
// welcome now equals "hello"
```

> NOTE
>
> You can use the `insert(_:at:)`, `insert(contentsOf:at:)`, `remove(at:)`, and `removeSubrange(_:)` methods on any type that conforms to the `RangeReplaceableCollection` protocol. This includes `String`, as shown here, as well as collection types such as `Array`, `Dictionary`, and `Set`.

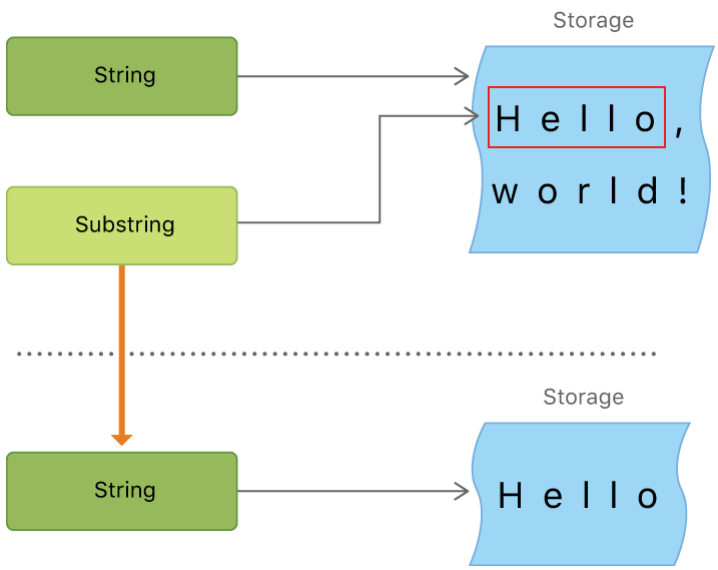
Substrings

When you get a substring from a string—for example, using a subscript or a method like `prefix(_:)`—the result is an instance of `Substring`, not another string. Substrings in Swift have most of the same methods as strings, which means you can work with substrings the same way you work with strings. However, unlike strings, you use substrings for only a short amount of time while performing actions on a string. When you're ready to store the result for a longer time, you convert the substring to an instance of `String`. For example:

```
let greeting = "Hello, world!"
let index = greeting.firstIndex(of: ",") ?? greeting.endIndex
let beginning = greeting[..<index]
// beginning is "Hello"

// Convert the result to a String for long-term storage.
let newString = String(beginning)
```

Like strings, each substring has a region of memory where the characters that make up the substring are stored. The difference between strings and substrings is that, as a performance optimization, a substring can reuse part of the memory that's used to store the original string, or part of the memory that's used to store another substring. (Strings have a similar optimization, but if two strings share memory, they're equal.) This performance optimization means you don't have to pay the performance cost of copying memory until you modify either the string or substring. As mentioned above, substrings aren't suitable for long-term storage—because they reuse the storage of the original string, the entire original string must be kept in memory as long as any of its substrings are being used.

In the example above, `greeting` is a string, which means it has a region of memory where the characters that make up the string are stored. Because `beginning` is a substring of `greeting`, it reuses the memory that `greeting` uses. In contrast, `newString` is a string—when it's created from the substring, it has its own storage. The figure below shows these relationships:



> NOTE
>
> Both `String` and `Substring` conform to the `StringProtocol` protocol, which means it's often convenient for string-manipulation functions to accept a `StringProtocol` value. You can call such functions with either a `String` or `Substring` value.

Comparing Strings

Swift provides three ways to compare textual values: string and character equality, prefix equality, and suffix equality.

String and Character Equality

String and character equality is checked with the "equal to" operator (`==`) and the "not equal to" operator (`!=`), as described in Comparison Operators:

```swift
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if quotation == sameQuotation {
    print("These two strings are considered equal")
}
// Prints "These two strings are considered equal"
```

Two `String` values (or two `Character` values) are considered equal if their extended grapheme clusters are *canonically equivalent*. Extended grapheme clusters are canonically equivalent if they have the same linguistic meaning and appearance, even if they're composed from different Unicode scalars behind the scenes.

For example, `LATIN SMALL LETTER E WITH ACUTE` (`U+00E9`) is canonically equivalent to `LATIN SMALL LETTER E` (`U+0065`) followed by `COMBINING ACUTE ACCENT` (`U+0301`). Both of these extended grapheme clusters are valid ways to represent the character `é`, and so they're considered to be canonically equivalent:

```
// "Voulez-vous un café?" using LATIN SMALL LETTER E WITH ACUTE
let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"

// "Voulez-vous un café?" using LATIN SMALL LETTER E and COMBINING ACUTE ACCENT
let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"

if eAcuteQuestion == combinedEAcuteQuestion {
    print("These two strings are considered equal")
}
// Prints "These two strings are considered equal"
```

Conversely, `LATIN CAPITAL LETTER A` (`U+0041`, or `"A"`), as used in English, is *not* equivalent to `CYRILLIC CAPITAL LETTER A` (`U+0410`, or `"А"`), as used in Russian. The characters are visually similar, but don't have the same linguistic meaning:

```
let latinCapitalLetterA: Character = "\u{41}"

let cyrillicCapitalLetterA: Character = "\u{0410}"

if latinCapitalLetterA != cyrillicCapitalLetterA {
    print("These two characters aren't equivalent.")
}
// Prints "These two characters aren't equivalent."
```

> NOTE
> String and character comparisons in Swift aren't locale-sensitive.

Prefix and Suffix Equality

To check whether a string has a particular string prefix or suffix, call the string's `hasPrefix(_:)` and `hasSuffix(_:)` methods, both of which take a single argument of type `String` and return a Boolean value.

The examples below consider an array of strings representing the scene locations from the first two acts of Shakespeare's *Romeo and Juliet*:

```
let romeoAndJuliet = [
    "Act 1 Scene 1: Verona, A public place",
    "Act 1 Scene 2: Capulet's mansion",
    "Act 1 Scene 3: A room in Capulet's mansion",
    "Act 1 Scene 4: A street outside Capulet's mansion",
    "Act 1 Scene 5: The Great Hall in Capulet's mansion",
    "Act 2 Scene 1: Outside Capulet's mansion",
    "Act 2 Scene 2: Capulet's orchard",
    "Act 2 Scene 3: Outside Friar Lawrence's cell",
    "Act 2 Scene 4: A street in Verona",
    "Act 2 Scene 5: Capulet's mansion",
    "Act 2 Scene 6: Friar Lawrence's cell"
]
```

You can use the `hasPrefix(_:)` method with the `romeoAndJuliet` array to count the number of scenes in Act 1 of the play:

```
var act1SceneCount = 0
for scene in romeoAndJuliet {
    if scene.hasPrefix("Act 1 ") {
        act1SceneCount += 1
    }
}
print("There are \(act1SceneCount) scenes in Act 1")
// Prints "There are 5 scenes in Act 1"
```

Similarly, use the `hasSuffix(_:)` method to count the number of scenes that take place in or around Capulet's mansion and Friar Lawrence's cell:

```
var mansionCount = 0
var cellCount = 0
for scene in romeoAndJuliet {
    if scene.hasSuffix("Capulet's mansion") {
        mansionCount += 1
    } else if scene.hasSuffix("Friar Lawrence's cell") {
        cellCount += 1
    }
}
print("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
// Prints "6 mansion scenes; 2 cell scenes"
```

> NOTE
>
> The `hasPrefix(_:)` and `hasSuffix(_:)` methods perform a character-by-character canonical equivalence comparison between the extended grapheme clusters in each string, as described in [String and Character Equality](#).

Unicode Representations of Strings

When a Unicode string is written to a text file or some other storage, the Unicode scalars in that string are encoded in one of several Unicode-defined *encoding forms*. Each form encodes the string in small chunks known as *code units*. These include the UTF-8 encoding form (which encodes a string as 8-bit code units), the UTF-16 encoding form (which encodes a string as

16-bit code units), and the UTF-32 encoding form (which encodes a string as 32-bit code units).

Swift provides several different ways to access Unicode representations of strings. You can iterate over the string with a `for-in` statement, to access its individual `Character` values as Unicode extended grapheme clusters. This process is described in [Working with Characters](#).

Alternatively, access a `String` value in one of three other Unicode-compliant representations:

- A collection of UTF-8 code units (accessed with the string's `utf8` property)
- A collection of UTF-16 code units (accessed with the string's `utf16` property)
- A collection of 21-bit Unicode scalar values, equivalent to the string's UTF-32 encoding form (accessed with the string's `unicodeScalars` property)

Each example below shows a different representation of the following string, which is made up of the characters `D`, `o`, `g`, `‼` (`DOUBLE EXCLAMATION MARK`, or Unicode scalar `U+203C`), and the 🐶 character (`DOG FACE`, or Unicode scalar `U+1F436`):

```
let dogString = "Dog‼🐶"
```

UTF-8 Representation

You can access a UTF-8 representation of a `String` by iterating over its `utf8` property. This property is of type `String.UTF8View`, which is a collection of unsigned 8-bit (`UInt8`) values, one for each byte in the string's UTF-8 representation:

Character	D U+0044	o U+006F	g U+0067	!! U+203C			🐶 U+1F436			
UTF-8 Code Unit	68	111	103	226	128	188	240	159	144	182
Position	0	1	2	3	4	5	6	7	8	9

```
for codeUnit in dogString.utf8 {
    print("\(codeUnit) ", terminator: "")
}
print("")
// Prints "68 111 103 226 128 188 240 159 144 182 "
```

In the example above, the first three decimal `codeUnit` values (`68`, `111`, `103`) represent the characters `D`, `o`, and `g`, whose UTF-8 representation is the same as their ASCII representation. The next three decimal `codeUnit` values (`226`, `128`, `188`) are a three-byte UTF-8 representation of the `DOUBLE EXCLAMATION MARK` character. The last four `codeUnit` values (`240`, `159`, `144`, `182`) are a four-byte UTF-8 representation of the `DOG FACE` character.

UTF-16 Representation

You can access a UTF-16 representation of a `String` by iterating over its `utf16` property. This property is of type `String.UTF16View`, which is a collection of unsigned 16-bit (`UInt16`) values, one for each 16-bit code unit in the string's UTF-16 representation:

Character	D U+0044	o U+006F	g U+0067	!! U+203C	🐶 U+1F436	
UTF-16 Code Unit	68	111	103	8252	55357	56374
Position	0	1	2	3	4	5

```
for codeUnit in dogString.utf16 {
    print("\(codeUnit) ", terminator: "")
}
print("")
// Prints "68 111 103 8252 55357 56374 "
```

Again, the first three `codeUnit` values (`68`, `111`, `103`) represent the characters `D`, `o`, and `g`, whose UTF-16 code units have the same values as in the string's UTF-8 representation (because these Unicode scalars represent ASCII characters).

The fourth `codeUnit` value (`8252`) is a decimal equivalent of the hexadecimal value `203C`, which represents the Unicode scalar `U+203C` for the `DOUBLE EXCLAMATION MARK` character. This character can be represented as a single code unit in UTF-16.

The fifth and sixth `codeUnit` values (`55357` and `56374`) are a UTF-16 surrogate pair representation of the `DOG FACE` character. These values are a high-surrogate value of `U+D83D` (decimal value `55357`) and a low-surrogate value of `U+DC36` (decimal value `56374`).

Unicode Scalar Representation

You can access a Unicode scalar representation of a `String` value by iterating over its `unicodeScalars` property. This property is of type `UnicodeScalarView`, which is a collection of values of type `UnicodeScalar`.

Each `UnicodeScalar` has a `value` property that returns the scalar's 21-bit value, represented within a `UInt32` value:

Character	D U+0044	o U+006F	g U+0067	!! U+203C	🐶 U+1F436
Unicode Scalar Code Unit	68	111	103	8252	128054
Position	0	1	2	3	4

```
for scalar in dogString.unicodeScalars {
    print("\(scalar.value) ", terminator: "")
}
print("")
// Prints "68 111 103 8252 128054 "
```

The `value` properties for the first three `UnicodeScalar` values (`68`, `111`, `103`) once again represent the characters `D`, `o`, and `g`.

The fourth `codeUnit` value (`8252`) is again a decimal equivalent of the hexadecimal value `203C`, which represents the Unicode scalar `U+203C` for the `DOUBLE EXCLAMATION MARK` character.

The `value` property of the fifth and final `UnicodeScalar`, `128054`, is a decimal equivalent of the hexadecimal value `1F436`, which represents the

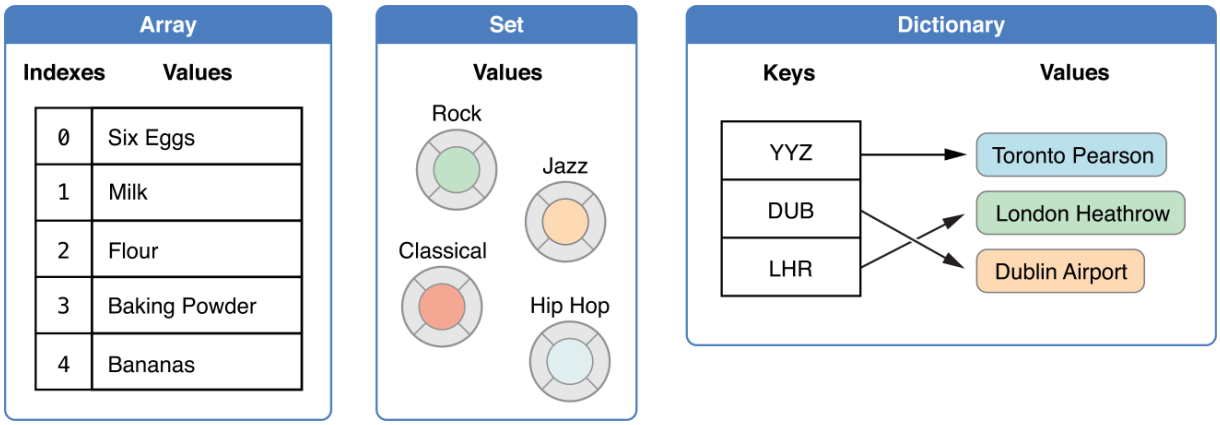
Unicode scalar `U+1F436` for the `DOG FACE` character.

As an alternative to querying their `value` properties, each `UnicodeScalar` value can also be used to construct a new `String` value, such as with string interpolation:

```
for scalar in dogString.unicodeScalars {
    print("\(scalar) ")
}
// D
// o
// g
// !!
// 🐶
```

Collection Types

Swift provides three primary *collection types*, known as arrays, sets, and dictionaries, for storing collections of values. Arrays are ordered collections of values. Sets are unordered collections of unique values. Dictionaries are unordered collections of key-value associations.



Arrays, sets, and dictionaries in Swift are always clear about the types of values and keys that they can store. This means that you can't insert a value of the wrong type into a collection by mistake. It also means you can be confident about the type of values you will retrieve from a collection.

> NOTE
>
> Swift's array, set, and dictionary types are implemented as *generic collections*. For more about generic types and collections, see Generics.

Mutability of Collections

If you create an array, a set, or a dictionary, and assign it to a variable, the collection that's created will be *mutable*. This means that you can change (or *mutate*) the collection after it's created by adding, removing, or changing

items in the collection. If you assign an array, a set, or a dictionary to a constant, that collection is *immutable*, and its size and contents can't be changed.

> NOTE
>
> It's good practice to create immutable collections in all cases where the collection doesn't need to change. Doing so makes it easier for you to reason about your code and enables the Swift compiler to optimize the performance of the collections you create.

Arrays

An *array* stores values of the same type in an ordered list. The same value can appear in an array multiple times at different positions.

> NOTE
>
> Swift's `Array` type is bridged to Foundation's `NSArray` class.
>
> For more information about using `Array` with Foundation and Cocoa, see [Bridging Between Array and NSArray](#).

Array Type Shorthand Syntax

The type of a Swift array is written in full as `Array<Element>`, where `Element` is the type of values the array is allowed to store. You can also write the type of an array in shorthand form as `[Element]`. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of an array.

Creating an Empty Array

You can create an empty array of a certain type using initializer syntax:

```
var someInts: [Int] = []
print("someInts is of type [Int] with \
     (someInts.count) items.")
// Prints "someInts is of type [Int] with 0 items."
```

Note that the type of the `someInts` variable is inferred to be `[Int]` from the type of the initializer.

Alternatively, if the context already provides type information, such as a function argument or an already typed variable or constant, you can create an empty array with an empty array literal, which is written as `[]` (an empty pair of square brackets):

```
someInts.append(3)
// someInts now contains 1 value of type Int
someInts = []
// someInts is now an empty array, but is still of
     type [Int]
```

Creating an Array with a Default Value

Swift's `Array` type also provides an initializer for creating an array of a certain size with all of its values set to the same default value. You pass this initializer a default value of the appropriate type (called `repeating`): and the number of times that value is repeated in the new array (called `count`):

```
var threeDoubles = Array(repeating: 0.0, count: 3)
// threeDoubles is of type [Double], and equals [0.0,
     0.0, 0.0]
```

Creating an Array by Adding Two Arrays Together

You can create a new array by adding together two existing arrays with compatible types with the addition operator (+). The new array's type is inferred from the type of the two arrays you add together:

```
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]

var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

Creating an Array with an Array Literal

You can also initialize an array with an *array literal*, which is a shorthand way to write one or more values as an array collection. An array literal is written as a list of values, separated by commas, surrounded by a pair of square brackets:

```
[value 1, value 2, value 3]
```

The example below creates an array called `shoppingList` to store `String` values:

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

The `shoppingList` variable is declared as "an array of string values", written as `[String]`. Because this particular array has specified a value type of `String`, it's allowed to store `String` values only. Here, the `shoppingList` array is initialized with two `String` values (`"Eggs"` and `"Milk"`), written within an array literal.

> NOTE
>
> The `shoppingList` array is declared as a variable (with the `var` introducer) and not a constant (with the `let` introducer) because more items are added to the shopping list in the examples below.

In this case, the array literal contains two `String` values and nothing else. This matches the type of the `shoppingList` variable's declaration (an array that can only contain `String` values), and so the assignment of the array literal is permitted as a way to initialize `shoppingList` with two initial items.

Thanks to Swift's type inference, you don't have to write the type of the array if you're initializing it with an array literal containing values of the same type. The initialization of `shoppingList` could have been written in a shorter form instead:

```
var shoppingList = ["Eggs", "Milk"]
```

Because all values in the array literal are of the same type, Swift can infer that `[String]` is the correct type to use for the `shoppingList` variable.

Accessing and Modifying an Array

You access and modify an array through its methods and properties, or by using subscript syntax.

To find out the number of items in an array, check its read-only `count` property:

```
print("The shopping list contains \
     (shoppingList.count) items.")
// Prints "The shopping list contains 2 items."
```

Use the Boolean `isEmpty` property as a shortcut for checking whether the `count` property is equal to `0`:

```
if shoppingList.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list isn't empty.")
}
// Prints "The shopping list isn't empty."
```

You can add a new item to the end of an array by calling the array's `append(_:)` method:

```
shoppingList.append("Flour")
// shoppingList now contains 3 items, and someone is
     making pancakes
```

Alternatively, append an array of one or more compatible items with the addition assignment operator (`+=`):

```
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
shoppingList += ["Chocolate Spread", "Cheese",
     "Butter"]
// shoppingList now contains 7 items
```

Retrieve a value from the array by using *subscript syntax*, passing the index of the value you want to retrieve within square brackets immediately after the name of the array:

```
var firstItem = shoppingList[0]
// firstItem is equal to "Eggs"
```

> NOTE
>
> The first item in the array has an index of `0`, not `1`. Arrays in Swift are always zero-indexed.

You can use subscript syntax to change an existing value at a given index:

```
shoppingList[0] = "Six eggs"
// the first item in the list is now equal to "Six
     eggs" rather than "Eggs"
```

When you use subscript syntax, the index you specify needs to be valid. For example, writing `shoppingList[shoppingList.count] = "Salt"` to try to append an item to the end of the array results in a runtime error.

You can also use subscript syntax to change a range of values at once, even if the replacement set of values has a different length than the range you are replacing. The following example replaces `"Chocolate Spread"`, `"Cheese"`, and `"Butter"` with `"Bananas"` and `"Apples"`:

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList now contains 6 items
```

To insert an item into the array at a specified index, call the array's `insert(_:at:)` method:

```
shoppingList.insert("Maple Syrup", at: 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list
```

This call to the `insert(_:at:)` method inserts a new item with a value of `"Maple Syrup"` at the very beginning of the shopping list, indicated by an index of `0`.

Similarly, you remove an item from the array with the `remove(at:)` method. This method removes the item at the specified index and returns the removed item (although you can ignore the returned value if you don't need it):

```
let mapleSyrup = shoppingList.remove(at: 0)
// the item that was at index 0 has just been removed
// shoppingList now contains 6 items, and no Maple
     Syrup
// the mapleSyrup constant is now equal to the
     removed "Maple Syrup" string
```

> NOTE
>
> If you try to access or modify a value for an index that's outside of an array's existing bounds, you will trigger a runtime error. You can check that an index is valid before using it by comparing it to the array's `count` property. The largest valid index in an array is `count - 1` because arrays are indexed from zero—however, when `count` is `0` (meaning the array is empty), there are no valid indexes.

Any gaps in an array are closed when an item is removed, and so the value at index `0` is once again equal to `"Six eggs"`:

```
firstItem = shoppingList[0]
// firstItem is now equal to "Six eggs"
```

If you want to remove the final item from an array, use the `removeLast()` method rather than the `remove(at:)` method to avoid the need to query the array's `count` property. Like the `remove(at:)` method, `removeLast()` returns the removed item:

```
let apples = shoppingList.removeLast()
// the last item in the array has just been removed
// shoppingList now contains 5 items, and no apples
// the apples constant is now equal to the removed
     "Apples" string
```

Iterating Over an Array

You can iterate over the entire set of values in an array with the `for-in` loop:

```
for item in shoppingList {
    print(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

If you need the integer index of each item as well as its value, use the `enumerated()` method to iterate over the array instead. For each item in the array, the `enumerated()` method returns a tuple composed of an integer and the item. The integers start at zero and count up by one for each item; if you enumerate over a whole array, these integers match the items' indices. You

can decompose the tuple into temporary constants or variables as part of the iteration:

```
for (index, value) in shoppingList.enumerated() {
    print("Item \(index + 1): \(value)")
}
// Item 1: Six eggs
// Item 2: Milk
// Item 3: Flour
// Item 4: Baking Powder
// Item 5: Bananas
```

For more about the `for-in` loop, see For-In Loops.

Sets

A *set* stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items isn't important, or when you need to ensure that an item only appears once.

> NOTE
>
> Swift's `Set` type is bridged to Foundation's `NSSet` class.
>
> For more information about using `Set` with Foundation and Cocoa, see Bridging Between Set and NSSet.

Hash Values for Set Types

A type must be *hashable* in order to be stored in a set—that is, the type must provide a way to compute a *hash value* for itself. A hash value is an `Int`

value that's the same for all objects that compare equally, such that if `a == b`, the hash value of `a` is equal to the hash value of `b`.

All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default, and can be used as set value types or dictionary key types. Enumeration case values without associated values (as described in Enumerations) are also hashable by default.

> NOTE
>
> You can use your own custom types as set value types or dictionary key types by making them conform to the `Hashable` protocol from the Swift standard library. For information about implementing the required `hash(into:)` method, see `Hashable`. For information about conforming to protocols, see Protocols.

Set Type Syntax

The type of a Swift set is written as `Set<Element>`, where `Element` is the type that the set is allowed to store. Unlike arrays, sets don't have an equivalent shorthand form.

Creating and Initializing an Empty Set

You can create an empty set of a certain type using initializer syntax:

```
var letters = Set<Character>()
print("letters is of type Set<Character> with \
      (letters.count) items.")
// Prints "letters is of type Set<Character> with 0
      items."
```

> NOTE
>
> The type of the `letters` variable is inferred to be `Set<Character>`, from the type of the initializer.

Alternatively, if the context already provides type information, such as a function argument or an already typed variable or constant, you can create an empty set with an empty array literal:

```
letters.insert("a")
// letters now contains 1 value of type Character
letters = []
// letters is now an empty set, but is still of type Set<Character>
```

Creating a Set with an Array Literal

You can also initialize a set with an array literal, as a shorthand way to write one or more values as a set collection.

The example below creates a set called `favoriteGenres` to store `String` values:

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres has been initialized with three initial items
```

The `favoriteGenres` variable is declared as "a set of `String` values", written as `Set<String>`. Because this particular set has specified a value type of `String`, it's *only* allowed to store `String` values. Here, the

`favoriteGenres` set is initialized with three `String` values (`"Rock"`, `"Classical"`, and `"Hip hop"`), written within an array literal.

> NOTE
>
> The `favoriteGenres` set is declared as a variable (with the `var` introducer) and not a constant (with the `let` introducer) because items are added and removed in the examples below.

A set type can't be inferred from an array literal alone, so the type `Set` must be explicitly declared. However, because of Swift's type inference, you don't have to write the type of the set's elements if you're initializing it with an array literal that contains values of just one type. The initialization of `favoriteGenres` could have been written in a shorter form instead:

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip
     hop"]
```

Because all values in the array literal are of the same type, Swift can infer that `Set<String>` is the correct type to use for the `favoriteGenres` variable.

Accessing and Modifying a Set

You access and modify a set through its methods and properties.

To find out the number of items in a set, check its read-only `count` property:

```
1 print("I have \(favoriteGenres.count) favorite music
     genres.")
2 // Prints "I have 3 favorite music genres."
```

Use the Boolean `isEmpty` property as a shortcut for checking whether the `count` property is equal to `0`:

```
if favoriteGenres.isEmpty {
    print("As far as music goes, I'm not picky.")
} else {
    print("I have particular music preferences.")
}
// Prints "I have particular music preferences."
```

You can add a new item into a set by calling the set's `insert(_:)` method:

```
favoriteGenres.insert("Jazz")
// favoriteGenres now contains 4 items
```

You can remove an item from a set by calling the set's `remove(_:)` method, which removes the item if it's a member of the set, and returns the removed value, or returns `nil` if the set didn't contain it. Alternatively, all items in a set can be removed with its `removeAll()` method.

```
if let removedGenre = favoriteGenres.remove("Rock") {
    print("\(removedGenre)? I'm over it.")
} else {
    print("I never much cared for that.")
}
// Prints "Rock? I'm over it."
```

To check whether a set contains a particular item, use the `contains(_:)` method.

```swift
if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
} else {
    print("It's too funky in here.")
}
// Prints "It's too funky in here."
```

Iterating Over a Set

You can iterate over the values in a set with a `for-in` loop.

```swift
for genre in favoriteGenres {
    print("\(genre)")
}
// Classical
// Jazz
// Hip hop
```

For more about the `for-in` loop, see [For-In Loops].

Swift's `Set` type doesn't have a defined ordering. To iterate over the values of a set in a specific order, use the `sorted()` method, which returns the set's elements as an array sorted using the < operator.

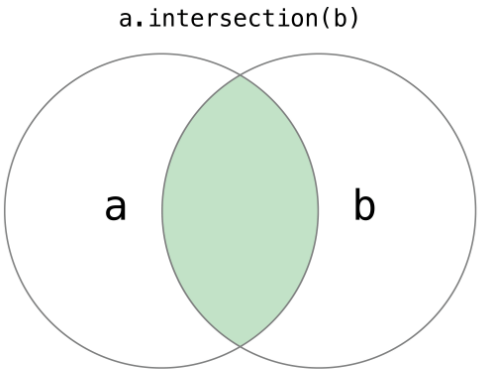
```swift
for genre in favoriteGenres.sorted() {
    print("\(genre)")
}
// Classical
// Hip hop
// Jazz
```

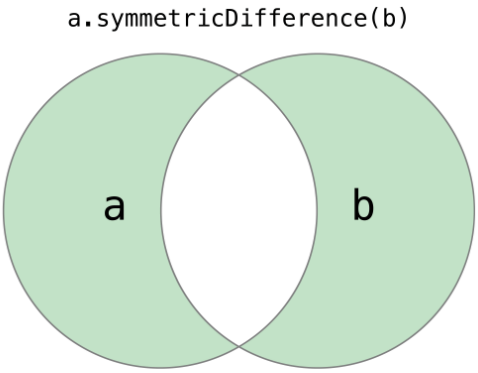
Performing Set Operations

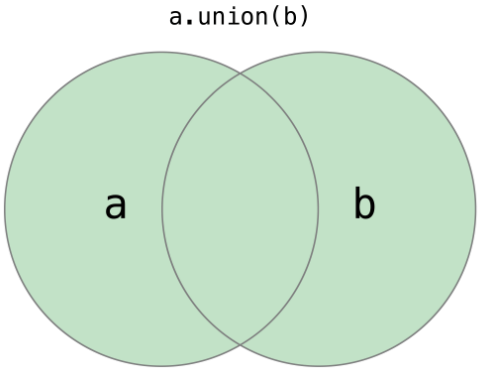
You can efficiently perform fundamental set operations, such as combining two sets together, determining which values two sets have in common, or determining whether two sets contain all, some, or none of the same values.

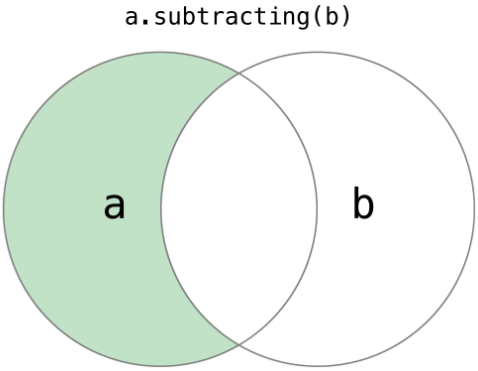
Fundamental Set Operations

The illustration below depicts two sets—`a` and `b`—with the results of various set operations represented by the shaded regions.







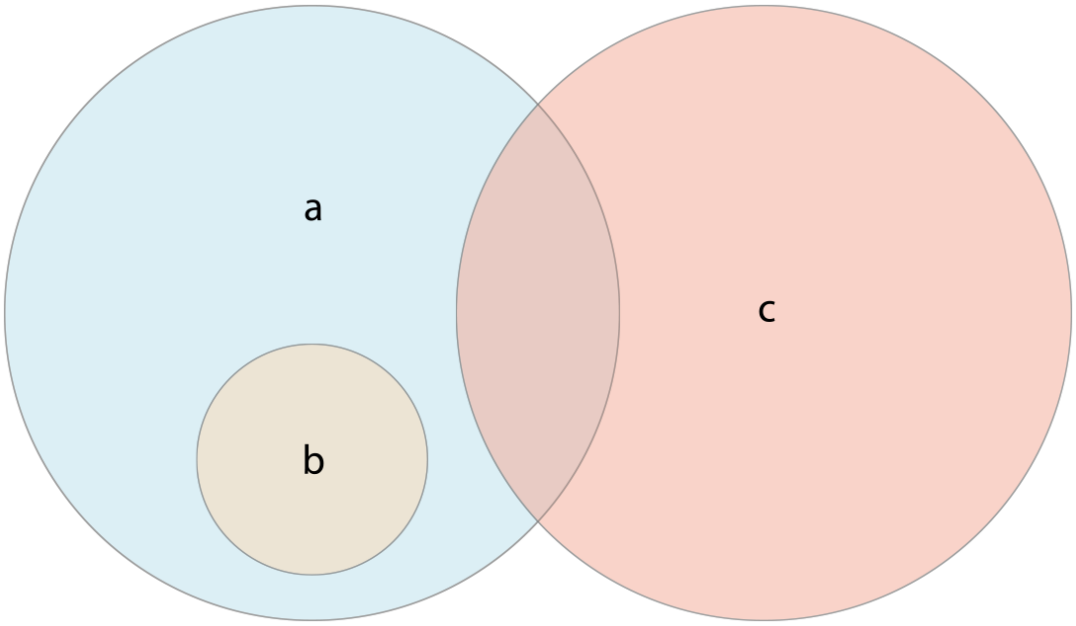


- Use the `intersection(_:)` method to create a new set with only the values common to both sets.
- Use the `symmetricDifference(_:)` method to create a new set with values in either set, but not both.
- Use the `union(_:)` method to create a new set with all of the values in both sets.
- Use the `subtracting(_:)` method to create a new set with values not in the specified set.

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```

Set Membership and Equality

The illustration below depicts three sets—`a`, `b` and `c`—with overlapping regions representing elements shared among sets. Set `a` is a *superset* of set `b`, because `a` contains all elements in `b`. Conversely, set `b` is a *subset* of set `a`, because all elements in `b` are also contained by `a`. Set `b` and set `c` are *disjoint* with one another, because they share no elements in common.



- Use the “is equal” operator (==) to determine whether two sets contain all of the same values.
- Use the `isSubset(of:)` method to determine whether all of the values of a set are contained in the specified set.
- Use the `isSuperset(of:)` method to determine whether a set contains all of the values in a specified set.
- Use the `isStrictSubset(of:)` or `isStrictSuperset(of:)` methods to determine whether a set is a subset or superset, but not equal to, a specified set.
- Use the `isDisjoint(with:)` method to determine whether two sets have no values in common.

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐷", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐌", "🐭"]

houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
farmAnimals.isDisjoint(with: cityAnimals)
// true
```

Dictionaries

A *dictionary* stores associations between keys of the same type and values of the same type in a collection with no defined ordering. Each value is associated with a unique *key*, which acts as an identifier for that value within the dictionary. Unlike items in an array, items in a dictionary don't have a specified order. You use a dictionary when you need to look up values based on their identifier, in much the same way that a real-world dictionary is used to look up the definition for a particular word.

> NOTE
>
> Swift's `Dictionary` type is bridged to Foundation's `NSDictionary` class.
>
> For more information about using `Dictionary` with Foundation and Cocoa, see Bridging Between Dictionary and NSDictionary.

Dictionary Type Shorthand Syntax

The type of a Swift dictionary is written in full as `Dictionary<Key, Value>`, where `Key` is the type of value that can be used as a dictionary key, and `Value` is the type of value that the dictionary stores for those keys.

> NOTE
> A dictionary `Key` type must conform to the `Hashable` protocol, like a set's value type.

You can also write the type of a dictionary in shorthand form as `[Key: Value]`. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of a dictionary.

Creating an Empty Dictionary

As with arrays, you can create an empty `Dictionary` of a certain type by using initializer syntax:

```
var namesOfIntegers: [Int: String] = [:]
// namesOfIntegers is an empty [Int: String] dictionary
```

This example creates an empty dictionary of type `[Int: String]` to store human-readable names of integer values. Its keys are of type `Int`, and its values are of type `String`.

If the context already provides type information, you can create an empty dictionary with an empty dictionary literal, which is written as `[:]` (a colon inside a pair of square brackets):

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers now contains 1 key-value pair
namesOfIntegers = [:]
// namesOfIntegers is once again an empty dictionary
     of type [Int: String]
```

Creating a Dictionary with a Dictionary Literal

You can also initialize a dictionary with a *dictionary literal*, which has a similar syntax to the array literal seen earlier. A dictionary literal is a shorthand way to write one or more key-value pairs as a `Dictionary` collection.

A *key-value pair* is a combination of a key and a value. In a dictionary literal, the key and value in each key-value pair are separated by a colon. The key-value pairs are written as a list, separated by commas, surrounded by a pair of square brackets:

```
[key 1: value 1, key 2: value 2, key 3:
     value 3]
```

The example below creates a dictionary to store the names of international airports. In this dictionary, the keys are three-letter International Air Transport Association codes, and the values are airport names:

```
var airports: [String: String] = ["YYZ": "Toronto
     Pearson", "DUB": "Dublin"]
```

The `airports` dictionary is declared as having a type of `[String: String]`, which means "a `Dictionary` whose keys are of type `String`, and whose values are also of type `String`".

> NOTE
>
> The `airports` dictionary is declared as a variable (with the `var` introducer), and not a constant (with the `let` introducer), because more airports are added to the dictionary in the examples below.

The `airports` dictionary is initialized with a dictionary literal containing two key-value pairs. The first pair has a key of `"YYZ"` and a value of `"Toronto Pearson"`. The second pair has a key of `"DUB"` and a value of `"Dublin"`.

This dictionary literal contains two `String: String` pairs. This key-value type matches the type of the `airports` variable declaration (a dictionary with only `String` keys, and only `String` values), and so the assignment of the dictionary literal is permitted as a way to initialize the `airports` dictionary with two initial items.

As with arrays, you don't have to write the type of the dictionary if you're initializing it with a dictionary literal whose keys and values have consistent types. The initialization of `airports` could have been written in a shorter form instead:

```
var airports = ["YYZ": "Toronto Pearson", "DUB":
     "Dublin"]
```

Because all keys in the literal are of the same type as each other, and likewise all values are of the same type as each other, Swift can infer that `[String: String]` is the correct type to use for the `airports` dictionary.

Accessing and Modifying a Dictionary

You access and modify a dictionary through its methods and properties, or by using subscript syntax.

As with an array, you find out the number of items in a `Dictionary` by checking its read-only `count` property:

```
print("The airports dictionary contains \
     (airports.count) items.")
// Prints "The airports dictionary contains 2 items."
```

Use the Boolean `isEmpty` property as a shortcut for checking whether the `count` property is equal to `0`:

```
if airports.isEmpty {
    print("The airports dictionary is empty.")
} else {
    print("The airports dictionary isn't empty.")
}
// Prints "The airports dictionary isn't empty."
```

You can add a new item to a dictionary with subscript syntax. Use a new key of the appropriate type as the subscript index, and assign a new value of the appropriate type:

```
airports["LHR"] = "London"
// the airports dictionary now contains 3 items
```

You can also use subscript syntax to change the value associated with a particular key:

```
airports["LHR"] = "London Heathrow"
// the value for "LHR" has been changed to "London
     Heathrow"
```

As an alternative to subscripting, use a dictionary's `updateValue(_:forKey:)` method to set or update the value for a particular key. Like the subscript examples above, the `updateValue(_:forKey:)` method sets a value for a key if none exists, or updates the value if that key already exists. Unlike a subscript, however, the `updateValue(_:forKey:)` method returns the *old* value after performing an update. This enables you to check whether or not an update took place.

The `updateValue(_:forKey:)` method returns an optional value of the dictionary's value type. For a dictionary that stores `String` values, for example, the method returns a value of type `String?`, or "optional `String`". This optional value contains the old value for that key if one existed before the update, or `nil` if no value existed:

```
if let oldValue = airports.updateValue("Dublin
     Airport", forKey: "DUB") {
    print("The old value for DUB was \(oldValue).")
}
// Prints "The old value for DUB was Dublin."
```

You can also use subscript syntax to retrieve a value from the dictionary for a particular key. Because it's possible to request a key for which no value exists, a dictionary's subscript returns an optional value of the dictionary's value type. If the dictionary contains a value for the requested key, the subscript returns an optional value containing the existing value for that key. Otherwise, the subscript returns `nil`:

```
if let airportName = airports["DUB"] {
    print("The name of the airport is \(airportName).")
} else {
    print("That airport isn't in the airports dictionary.")
}
// Prints "The name of the airport is Dublin Airport."
```

You can use subscript syntax to remove a key-value pair from a dictionary by assigning a value of `nil` for that key:

```
airports["APL"] = "Apple International"
// "Apple International" isn't the real airport for APL, so delete it
airports["APL"] = nil
// APL has now been removed from the dictionary
```

Alternatively, remove a key-value pair from a dictionary with the `removeValue(forKey:)` method. This method removes the key-value pair if it exists and returns the removed value, or returns `nil` if no value existed:

```
if let removedValue = airports.removeValue(forKey: "DUB") {
    print("The removed airport's name is \(removedValue).")
} else {
    print("The airports dictionary doesn't contain a value for DUB.")
}
// Prints "The removed airport's name is Dublin Airport."
```

Iterating Over a Dictionary

You can iterate over the key-value pairs in a dictionary with a `for-in` loop. Each item in the dictionary is returned as a `(key, value)` tuple, and you can decompose the tuple's members into temporary constants or variables as part of the iteration:

```
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
// LHR: London Heathrow
// YYZ: Toronto Pearson
```

For more about the `for-in` loop, see [For-In Loops](#).

You can also retrieve an iterable collection of a dictionary's keys or values by accessing its `keys` and `values` properties:

```
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}
// Airport code: LHR
// Airport code: YYZ

for airportName in airports.values {
    print("Airport name: \(airportName)")
}
// Airport name: London Heathrow
// Airport name: Toronto Pearson
```

If you need to use a dictionary's keys or values with an API that takes an `Array` instance, initialize a new array with the `keys` or `values` property:

```
let airportCodes = [String](airports.keys)
// airportCodes is ["LHR", "YYZ"]

let airportNames = [String](airports.values)
// airportNames is ["London Heathrow", "Toronto
     Pearson"]
```

Swift's `Dictionary` type doesn't have a defined ordering. To iterate over the keys or values of a dictionary in a specific order, use the `sorted()` method on its `keys` or `values` property.

Control Flow

Swift provides a variety of control flow statements. These include `while` loops to perform a task multiple times; `if`, `guard`, and `switch` statements to execute different branches of code based on certain conditions; and statements such as `break` and `continue` to transfer the flow of execution to another point in your code.

Swift also provides a `for-in` loop that makes it easy to iterate over arrays, dictionaries, ranges, strings, and other sequences.

Swift's `switch` statement is considerably more powerful than its counterpart in many C-like languages. Cases can match many different patterns, including interval matches, tuples, and casts to a specific type. Matched values in a `switch` case can be bound to temporary constants or variables for use within the case's body, and complex matching conditions can be expressed with a `where` clause for each case.

For-In Loops

You use the `for-in` loop to iterate over a sequence, such as items in an array, ranges of numbers, or characters in a string.

This example uses a `for-in` loop to iterate over the items in an array:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

You can also iterate over a dictionary to access its key-value pairs. Each item in the dictionary is returned as a `(key, value)` tuple when the dictionary is iterated, and you

can decompose the `(key, value)` tuple's members as explicitly named constants for use within the body of the `for-in` loop. In the code example below, the dictionary's keys are decomposed into a constant called `animalName`, and the dictionary's values are decomposed into a constant called `legCount`.

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
// cats have 4 legs
// ants have 6 legs
// spiders have 8 legs
```

The contents of a `Dictionary` are inherently unordered, and iterating over them doesn't guarantee the order in which they will be retrieved. In particular, the order you insert items into a `Dictionary` doesn't define the order they're iterated. For more about arrays and dictionaries, see Collection Types.

You can also use `for-in` loops with numeric ranges. This example prints the first few entries in a five-times table:

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

The sequence being iterated over is a range of numbers from `1` to `5`, inclusive, as indicated by the use of the closed range operator (`...`). The value of `index` is set to the first number in the range (`1`), and the statements inside the loop are executed. In this case, the loop contains only one statement, which prints an entry from the five-times table for the current value of `index`. After the statement is executed, the value of `index`

is updated to contain the second value in the range (2), and the `print(_:separator:terminator:)` function is called again. This process continues until the end of the range is reached.

In the example above, `index` is a constant whose value is automatically set at the start of each iteration of the loop. As such, `index` doesn't have to be declared before it's used. It's implicitly declared simply by its inclusion in the loop declaration, without the need for a `let` declaration keyword.

If you don't need each value from a sequence, you can ignore the values by using an underscore in place of a variable name.

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
print("\(base) to the power of \(power) is \(answer)")
// Prints "3 to the power of 10 is 59049"
```

The example above calculates the value of one number to the power of another (in this case, `3` to the power of `10`). It multiplies a starting value of `1` (that is, `3` to the power of `0`) by `3`, ten times, using a closed range that starts with `1` and ends with `10`. For this calculation, the individual counter values each time through the loop are unnecessary—the code simply executes the loop the correct number of times. The underscore character (_) used in place of a loop variable causes the individual values to be ignored and doesn't provide access to the current value during each iteration of the loop.

In some situations, you might not want to use closed ranges, which include both endpoints. Consider drawing the tick marks for every minute on a watch face. You want to draw `60` tick marks, starting with the `0` minute. Use the half-open range operator (`..<`) to include the lower bound but not the upper bound. For more about ranges, see Range Operators.

```
let minutes = 60
for tickMark in 0..<minutes {
    // render the tick mark each minute (60 times)
}
```

Some users might want fewer tick marks in their UI. They could prefer one mark every `5` minutes instead. Use the `stride(from:to:by:)` function to skip the unwanted marks.

```
let minuteInterval = 5
for tickMark in stride(from: 0, to: minutes, by:
     minuteInterval) {
    // render the tick mark every 5 minutes (0, 5, 10, 15 ...
     45, 50, 55)
}
```

Closed ranges are also available, by using `stride(from:through:by:)` instead:

```
let hours = 12
let hourInterval = 3
for tickMark in stride(from: 3, through: hours, by:
     hourInterval) {
    // render the tick mark every 3 hours (3, 6, 9, 12)
}
```

The examples above use a `for-in` loop to iterate ranges, arrays, dictionaries, and strings. However, you can use this syntax to iterate *any* collection, including your own classes and collection types, as long as those types conform to the `Sequence` protocol.

While Loops

A `while` loop performs a set of statements until a condition becomes `false`. These kinds of loops are best used when the number of iterations isn't known before the first iteration begins. Swift provides two kinds of `while` loops:

- `while` evaluates its condition at the start of each pass through the loop.
- `repeat-while` evaluates its condition at the end of each pass through the loop.

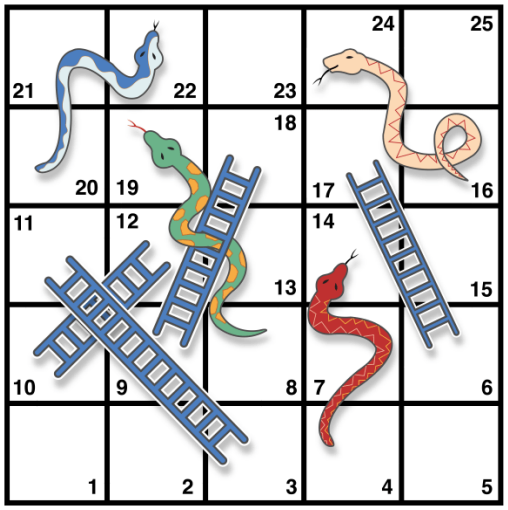
While

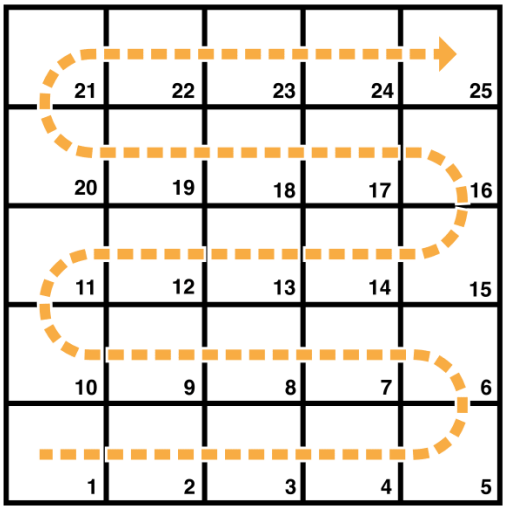
A `while` loop starts by evaluating a single condition. If the condition is `true`, a set of statements is repeated until the condition becomes `false`.

Here's the general form of a `while` loop:

```
while condition {
    statements
}
```

This example plays a simple game of *Snakes and Ladders* (also known as *Chutes and Ladders*):





The rules of the game are as follows:

- The board has 25 squares, and the aim is to land on or beyond square 25.

- The player's starting square is "square zero", which is just off the bottom-left corner of the board.
- Each turn, you roll a six-sided dice and move by that number of squares, following the horizontal path indicated by the dotted arrow above.
- If your turn ends at the bottom of a ladder, you move up that ladder.
- If your turn ends at the head of a snake, you move down that snake.

The game board is represented by an array of `Int` values. Its size is based on a constant called `finalSquare`, which is used to initialize the array and also to check for a win condition later in the example. Because the players start off the board, on "square zero", the board is initialized with 26 zero `Int` values, not 25.

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
```

Some squares are then set to have more specific values for the snakes and ladders. Squares with a ladder base have a positive number to move you up the board, whereas squares with a snake head have a negative number to move you back down the board.

```
board[03] = +08; board[06] = +11; board[09] = +09; board[10]
      = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24]
      = -08
```

Square 3 contains the bottom of a ladder that moves you up to square 11. To represent this, `board[03]` is equal to `+08`, which is equivalent to an integer value of `8` (the difference between `3` and `11`). To align the values and statements, the unary plus operator (`+i`) is explicitly used with the unary minus operator (`-i`) and numbers lower than `10` are padded with zeros. (Neither stylistic technique is strictly necessary, but they lead to neater code.)

```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // roll the dice
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    // move by the rolled amount
    square += diceRoll
    if square < board.count {
        // if we're still on the board, move up or down for a snake or a ladder
        square += board[square]
    }
}
print("Game over!")
```

The example above uses a very simple approach to dice rolling. Instead of generating a random number, it starts with a `diceRoll` value of `0`. Each time through the `while` loop, `diceRoll` is incremented by one and is then checked to see whether it has become too large. Whenever this return value equals `7`, the dice roll has become too large and is reset to a value of `1`. The result is a sequence of `diceRoll` values that's always `1`, `2`, `3`, `4`, `5`, `6`, `1`, `2` and so on.

After rolling the dice, the player moves forward by `diceRoll` squares. It's possible that the dice roll may have moved the player beyond square 25, in which case the game is over. To cope with this scenario, the code checks that `square` is less than the `board` array's `count` property. If `square` is valid, the value stored in `board[square]` is added to the current `square` value to move the player up or down any ladders or snakes.

> NOTE
>
> If this check isn't performed, `board[square]` might try to access a value outside the bounds of the `board` array, which would trigger a runtime error.

The current `while` loop execution then ends, and the loop's condition is checked to see if the loop should be executed again. If the player has moved on or beyond square

number 25, the loop's condition evaluates to false and the game ends.

A while loop is appropriate in this case, because the length of the game isn't clear at the start of the while loop. Instead, the loop is executed until a particular condition is satisfied.

Repeat-While

The other variation of the while loop, known as the repeat-while loop, performs a single pass through the loop block first, *before* considering the loop's condition. It then continues to repeat the loop until the condition is false.

> NOTE
>
> The repeat-while loop in Swift is analogous to a do-while loop in other languages.

Here's the general form of a repeat-while loop:

```
repeat {
    statements
} while condition
```

Here's the *Snakes and Ladders* example again, written as a repeat-while loop rather than a while loop. The values of finalSquare, board, square, and diceRoll are initialized in exactly the same way as with a while loop.

```
1  let finalSquare = 25
2  var board = [Int](repeating: 0, count: finalSquare + 1)
3  board[03] = +08; board[06] = +11; board[09] = +09; board[10]
       = +02
4  board[14] = -10; board[19] = -11; board[22] = -02; board[24]
       = -08
5  var square = 0
6  var diceRoll = 0
```

In this version of the game, the *first* action in the loop is to check for a ladder or a snake. No ladder on the board takes the player straight to square 25, and so it isn't possible to win the game by moving up a ladder. Therefore, it's safe to check for a snake or a ladder as the first action in the loop.

At the start of the game, the player is on "square zero". `board[0]` always equals `0` and has no effect.

```
repeat {
    // move up or down for a snake or ladder
    square += board[square]
    // roll the dice
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    // move by the rolled amount
    square += diceRoll
} while square < finalSquare
print("Game over!")
```

After the code checks for snakes and ladders, the dice is rolled and the player is moved forward by `diceRoll` squares. The current loop execution then ends.

The loop's condition (`while square < finalSquare`) is the same as before, but this time it's not evaluated until the *end* of the first run through the loop. The structure of the `repeat-while` loop is better suited to this game than the `while` loop in the previous example. In the `repeat-while` loop above, `square += board[square]` is always executed *immediately after* the loop's `while` condition confirms that `square` is still on the board. This behavior removes the need for the array bounds check seen in the `while` loop version of the game described earlier.

Conditional Statements

It's often useful to execute different pieces of code based on certain conditions. You might want to run an extra piece of code when an error occurs, or to display a message

when a value becomes too high or too low. To do this, you make parts of your code *conditional*.

Swift provides two ways to add conditional branches to your code: the `if` statement and the `switch` statement. Typically, you use the `if` statement to evaluate simple conditions with only a few possible outcomes. The `switch` statement is better suited to more complex conditions with multiple possible permutations and is useful in situations where pattern matching can help select an appropriate code branch to execute.

If

In its simplest form, the `if` statement has a single `if` condition. It executes a set of statements only if that condition is `true`.

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
}
// Prints "It's very cold. Consider wearing a scarf."
```

The example above checks whether the temperature is less than or equal to 32 degrees Fahrenheit (the freezing point of water). If it is, a message is printed. Otherwise, no message is printed, and code execution continues after the `if` statement's closing brace.

The `if` statement can provide an alternative set of statements, known as an *else clause*, for situations when the `if` condition is `false`. These statements are indicated by the `else` keyword.

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// Prints "It's not that cold. Wear a t-shirt."
```

One of these two branches is always executed. Because the temperature has increased to `40` degrees Fahrenheit, it's no longer cold enough to advise wearing a scarf and so the `else` branch is triggered instead.

You can chain multiple `if` statements together to consider additional clauses.

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear
     sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// Prints "It's really warm. Don't forget to wear sunscreen."
```

Here, an additional `if` statement was added to respond to particularly warm temperatures. The final `else` clause remains, and it prints a response for any temperatures that are neither too warm nor too cold.

The final `else` clause is optional, however, and can be excluded if the set of conditions doesn't need to be complete.

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear
     sunscreen.")
}
```

Because the temperature is neither too cold nor too warm to trigger the `if` or `else if` conditions, no message is printed.

Switch

A `switch` statement considers a value and compares it against several possible matching patterns. It then executes an appropriate block of code, based on the first pattern that matches successfully. A `switch` statement provides an alternative to the `if` statement for responding to multiple potential states.

In its simplest form, a `switch` statement compares a value against one or more values of the same type.

```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,
     value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

Every `switch` statement consists of multiple possible *cases*, each of which begins with the `case` keyword. In addition to comparing against specific values, Swift provides several ways for each case to specify more complex matching patterns. These options are described later in this chapter.

Like the body of an `if` statement, each `case` is a separate branch of code execution. The `switch` statement determines which branch should be selected. This procedure is known as *switching* on the value that's being considered.

Every `switch` statement must be *exhaustive*. That is, every possible value of the type being considered must be matched by one of the `switch` cases. If it's not appropriate to provide a case for every possible value, you can define a default case to cover any values that aren't addressed explicitly. This default case is indicated by the `default` keyword, and must always appear last.

This example uses a `switch` statement to consider a single lowercase character called `someCharacter`:

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
    print("The first letter of the alphabet")
case "z":
    print("The last letter of the alphabet")
default:
    print("Some other character")
}
// Prints "The last letter of the alphabet"
```

The `switch` statement's first case matches the first letter of the English alphabet, `a`, and its second case matches the last letter, `z`. Because the `switch` must have a case for every possible character, not just every alphabetic character, this `switch` statement uses a `default` case to match all characters other than `a` and `z`. This provision ensures that the `switch` statement is exhaustive.

No Implicit Fallthrough

In contrast with `switch` statements in C and Objective-C, `switch` statements in Swift don't fall through the bottom of each case and into the next one by default. Instead, the entire `switch` statement finishes its execution as soon as the first matching `switch` case is completed, without requiring an explicit `break` statement. This makes the `switch` statement safer and easier to use than the one in C and avoids executing more than one `switch` case by mistake.

> NOTE
>
> Although `break` isn't required in Swift, you can use a `break` statement to match and ignore a particular case or to break out of a matched case before that case has completed its execution. For details, see Break in a Switch Statement.

The body of each case *must* contain at least one executable statement. It isn't valid to write the following code, because the first case is empty:

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a": // Invalid, the case has an empty body
case "A":
    print("The letter A")
default:
    print("Not the letter A")
}
// This will report a compile-time error.
```

Unlike a `switch` statement in C, this `switch` statement doesn't match both `"a"` and `"A"`. Rather, it reports a compile-time error that `case "a":` doesn't contain any executable statements. This approach avoids accidental fallthrough from one case to another and makes for safer code that's clearer in its intent.

To make a `switch` with a single case that matches both `"a"` and `"A"`, combine the two values into a compound case, separating the values with commas.

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a", "A":
    print("The letter A")
default:
    print("Not the letter A")
}
// Prints "The letter A"
```

For readability, a compound case can also be written over multiple lines. For more information about compound cases, see Compound Cases.

> NOTE
> To explicitly fall through at the end of a particular `switch` case, use the `fallthrough` keyword, as described in Fallthrough.

Interval Matching

Values in `switch` cases can be checked for their inclusion in an interval. This example uses number intervals to provide a natural-language count for numbers of any size:

```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
let naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")
// Prints "There are dozens of moons orbiting Saturn."
```

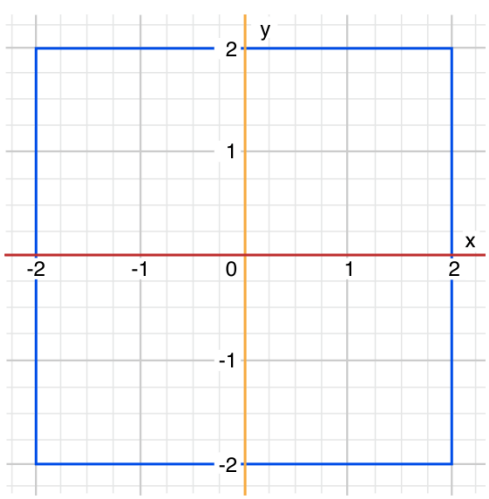
In the above example, `approximateCount` is evaluated in a `switch` statement. Each `case` compares that value to a number or interval. Because the value of `approximateCount` falls between 12 and 100, `naturalCount` is assigned the value `"dozens of"`, and execution is transferred out of the `switch` statement.

Tuples

You can use tuples to test multiple values in the same `switch` statement. Each element of the tuple can be tested against a different value or interval of values. Alternatively, use the underscore character (_), also known as the wildcard pattern, to match any possible value.

The example below takes an (x, y) point, expressed as a simple tuple of type `(Int, Int)`, and categorizes it on the graph that follows the example.

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
// Prints "(1, 1) is inside the box"
```



The `switch` statement determines whether the point is at the origin (0, 0), on the red x-axis, on the orange y-axis, inside the blue 4-by-4 box centered on the origin, or outside of the box.

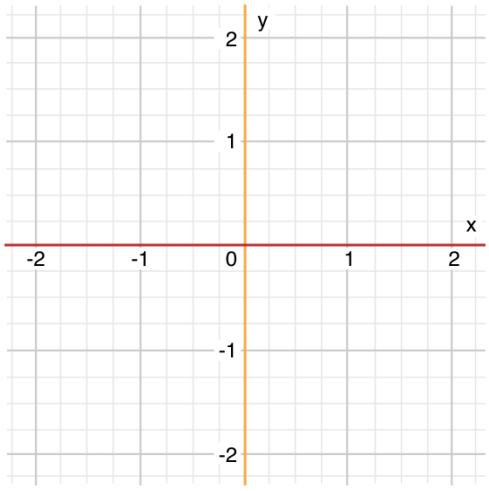
Unlike C, Swift allows multiple `switch` cases to consider the same value or values. In fact, the point (0, 0) could match all *four* of the cases in this example. However, if multiple matches are possible, the first matching case is always used. The point (0, 0) would match `case (0, 0)` first, and so all other matching cases would be ignored.

Value Bindings

A `switch` case can name the value or values it matches to temporary constants or variables, for use in the body of the case. This behavior is known as *value binding*, because the values are bound to temporary constants or variables within the case's body.

The example below takes an (x, y) point, expressed as a tuple of type `(Int, Int)`, and categorizes it on the graph that follows:

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
// Prints "on the x-axis with an x value of 2"
```



The `switch` statement determines whether the point is on the red x-axis, on the orange y-axis, or elsewhere (on neither axis).

The three `switch` cases declare placeholder constants `x` and `y`, which temporarily take on one or both tuple values from `anotherPoint`. The first case, `case (let x, 0)`, matches any point with a `y` value of `0` and assigns the point's `x` value to the temporary constant `x`. Similarly, the second case, `case (0, let y)`, matches any point with an `x` value of `0` and assigns the point's `y` value to the temporary constant `y`.

After the temporary constants are declared, they can be used within the case's code block. Here, they're used to print the categorization of the point.

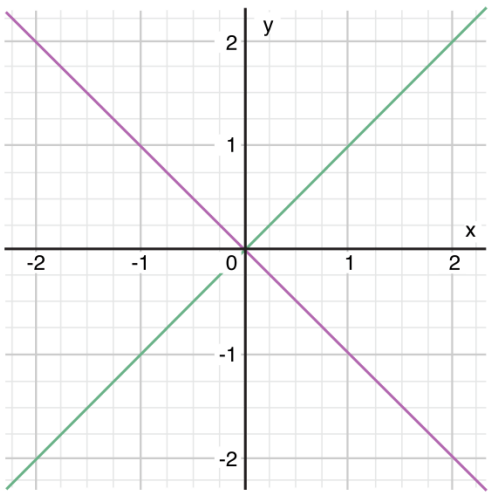
This `switch` statement doesn't have a `default` case. The final case, `case let (x, y)`, declares a tuple of two placeholder constants that can match any value. Because `anotherPoint` is always a tuple of two values, this case matches all possible remaining values, and a `default` case isn't needed to make the `switch` statement exhaustive.

Where

A `switch` case can use a `where` clause to check for additional conditions.

The example below categorizes an (x, y) point on the following graph:

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    print("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    print("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    print("(\(x), \(y)) is just some arbitrary point")
}
// Prints "(1, -1) is on the line x == -y"
```



The `switch` statement determines whether the point is on the green diagonal line where `x == y`, on the purple diagonal line where `x == -y`, or neither.

The three `switch` cases declare placeholder constants `x` and `y`, which temporarily take on the two tuple values from `yetAnotherPoint`. These constants are used as part of a `where` clause, to create a dynamic filter. The `switch` case matches the current value of `point` only if the `where` clause's condition evaluates to `true` for that value.

As in the previous example, the final case matches all possible remaining values, and so a `default` case isn't needed to make the `switch` statement exhaustive.

Compound Cases

Multiple switch cases that share the same body can be combined by writing several patterns after `case`, with a comma between each of the patterns. If any of the patterns match, then the case is considered to match. The patterns can be written over multiple lines if the list is long. For example:

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
     "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) isn't a vowel or a consonant")
}
// Prints "e is a vowel"
```

The `switch` statement's first case matches all five lowercase vowels in the English language. Similarly, its second case matches all lowercase English consonants. Finally, the `default` case matches any other character.

Compound cases can also include value bindings. All of the patterns of a compound case have to include the same set of value bindings, and each binding has to get a value of the same type from all of the patterns in the compound case. This ensures that, no matter which part of the compound case matched, the code in the body of the case can always access a value for the bindings and that the value always has the same type.

```
let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
case (let distance, 0), (0, let distance):
    print("On an axis, \(distance) from the origin")
default:
    print("Not on an axis")
}
// Prints "On an axis, 9 from the origin"
```

The `case` above has two patterns: `(let distance, 0)` matches points on the x-axis and `(0, let distance)` matches points on the y-axis. Both patterns include a binding for `distance` and `distance` is an integer in both patterns—which means that the code in the body of the `case` can always access a value for `distance`.

Control Transfer Statements

Control transfer statements change the order in which your code is executed, by transferring control from one piece of code to another. Swift has five control transfer statements:

- `continue`
- `break`
- `fallthrough`
- `return`
- `throw`

The `continue`, `break`, and `fallthrough` statements are described below. The `return` statement is described in Functions, and the `throw` statement is described in Propagating Errors Using Throwing Functions.

Continue
The `continue` statement tells a loop to stop what it's doing and start again at the beginning of the next iteration through the loop. It says "I am done with the current loop iteration" without leaving the loop altogether.

The following example removes all vowels and spaces from a lowercase string to create a cryptic puzzle phrase:

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
let charactersToRemove: [Character] = ["a", "e", "i", "o",
     "u", " "]
for character in puzzleInput {
    if charactersToRemove.contains(character) {
        continue
    }
    puzzleOutput.append(character)
}
print(puzzleOutput)
// Prints "grtmndsthnklk"
```

The code above calls the `continue` keyword whenever it matches a vowel or a space, causing the current iteration of the loop to end immediately and to jump straight to the start of the next iteration.

Break
The `break` statement ends execution of an entire control flow statement immediately. The `break` statement can be used inside a `switch` or loop statement when you want to terminate the execution of the `switch` or loop statement earlier than would otherwise be the case.

Break in a Loop Statement

When used inside a loop statement, `break` ends the loop's execution immediately and transfers control to the code after the loop's closing brace (`}`). No further code from the

current iteration of the loop is executed, and no further iterations of the loop are started.

Break in a Switch Statement

When used inside a `switch` statement, `break` causes the `switch` statement to end its execution immediately and to transfer control to the code after the `switch` statement's closing brace (`}`).

This behavior can be used to match and ignore one or more cases in a `switch` statement. Because Swift's `switch` statement is exhaustive and doesn't allow empty cases, it's sometimes necessary to deliberately match and ignore a case in order to make your intentions explicit. You do this by writing the `break` statement as the entire body of the case you want to ignore. When that case is matched by the `switch` statement, the `break` statement inside the case ends the `switch` statement's execution immediately.

> NOTE
>
> A `switch` case that contains only a comment is reported as a compile-time error. Comments aren't statements and don't cause a `switch` case to be ignored. Always use a `break` statement to ignore a `switch` case.

The following example switches on a `Character` value and determines whether it represents a number symbol in one of four languages. For brevity, multiple values are covered in a single `switch` case.

```
let numberSymbol: Character = "三"  // Chinese symbol for the number 3
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "๑":
    possibleIntegerValue = 1
case "2", "٢", "二", "๒":
    possibleIntegerValue = 2
case "3", "٣", "三", "๓":
    possibleIntegerValue = 3
case "4", "٤", "四", "๔":
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
    print("The integer value of \(numberSymbol) is \(integerValue).")
} else {
    print("An integer value couldn't be found for \(numberSymbol).")
}
// Prints "The integer value of 三 is 3."
```

This example checks `numberSymbol` to determine whether it's a Latin, Arabic, Chinese, or Thai symbol for the numbers `1` to `4`. If a match is found, one of the `switch` statement's cases sets an optional `Int?` variable called `possibleIntegerValue` to an appropriate integer value.

After the `switch` statement completes its execution, the example uses optional binding to determine whether a value was found. The `possibleIntegerValue` variable has an implicit initial value of `nil` by virtue of being an optional type, and so the optional

binding will succeed only if `possibleIntegerValue` was set to an actual value by one of the `switch` statement's first four cases.

Because it's not practical to list every possible `Character` value in the example above, a `default` case handles any characters that aren't matched. This `default` case doesn't need to perform any action, and so it's written with a single `break` statement as its body. As soon as the `default` case is matched, the `break` statement ends the `switch` statement's execution, and code execution continues from the `if let` statement.

Fallthrough

In Swift, `switch` statements don't fall through the bottom of each case and into the next one. That is, the entire `switch` statement completes its execution as soon as the first matching case is completed. By contrast, C requires you to insert an explicit `break` statement at the end of every `switch` case to prevent fallthrough. Avoiding default fallthrough means that Swift `switch` statements are much more concise and predictable than their counterparts in C, and thus they avoid executing multiple `switch` cases by mistake.

If you need C-style fallthrough behavior, you can opt in to this behavior on a case-by-case basis with the `fallthrough` keyword. The example below uses `fallthrough` to create a textual description of a number.

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
// Prints "The number 5 is a prime number, and also an
     integer."
```

This example declares a new `String` variable called `description` and assigns it an initial value. The function then considers the value of `integerToDescribe` using a `switch` statement. If the value of `integerToDescribe` is one of the prime numbers in the list, the function appends text to the end of `description`, to note that the number is prime. It then uses the `fallthrough` keyword to "fall into" the `default` case as well. The `default` case adds some extra text to the end of the description, and the `switch` statement is complete.

Unless the value of `integerToDescribe` is in the list of known prime numbers, it isn't matched by the first `switch` case at all. Because there are no other specific cases, `integerToDescribe` is matched by the `default` case.

After the `switch` statement has finished executing, the number's description is printed using the `print(_:separator:terminator:)` function. In this example, the number `5` is correctly identified as a prime number.

> NOTE
>
> The `fallthrough` keyword doesn't check the case conditions for the `switch` case that it causes execution to fall into. The `fallthrough` keyword simply causes code execution to move directly to the statements inside the next case (or `default` case) block, as in C's standard `switch` statement behavior.

Labeled Statements

In Swift, you can nest loops and conditional statements inside other loops and conditional statements to create complex control flow structures. However, loops and conditional statements can both use the `break` statement to end their execution prematurely. Therefore, it's sometimes useful to be explicit about which loop or conditional statement you want a `break` statement to terminate. Similarly, if you have multiple nested loops, it can be useful to be explicit about which loop the `continue` statement should affect.

To achieve these aims, you can mark a loop statement or conditional statement with a *statement label*. With a conditional statement, you can use a statement label with the `break` statement to end the execution of the labeled statement. With a loop statement, you can use a statement label with the `break` or `continue` statement to end or continue the execution of the labeled statement.

A labeled statement is indicated by placing a label on the same line as the statement's introducer keyword, followed by a colon. Here's an example of this syntax for a `while` loop, although the principle is the same for all loops and `switch` statements:

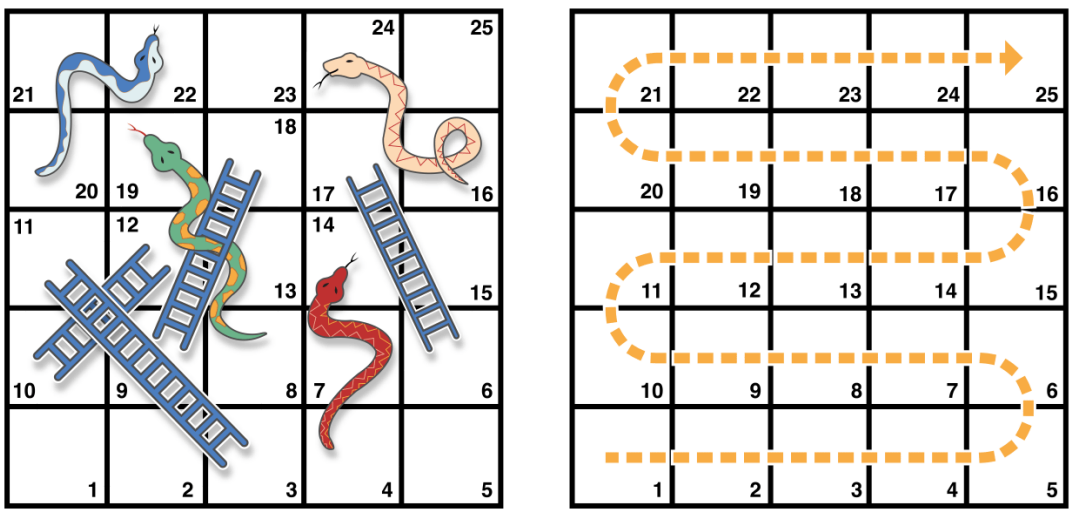
```
label name: while condition {
    statements
}
```

The following example uses the `break` and `continue` statements with a labeled `while` loop for an adapted version of the *Snakes and Ladders* game that you saw earlier in this chapter. This time around, the game has an extra rule:

- To win, you must land *exactly* on square 25.

If a particular dice roll would take you beyond square 25, you must roll again until you roll the exact number needed to land on square 25.

The game board is the same as before.



The values of `finalSquare`, `board`, `square`, and `diceRoll` are initialized in the same way as before:

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
board[03] = +08; board[06] = +11; board[09] = +09; board[10]
     = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24]
     = -08
var square = 0
var diceRoll = 0
```

This version of the game uses a `while` loop and a `switch` statement to implement the game's logic. The `while` loop has a statement label called `gameLoop` to indicate that it's the main game loop for the Snakes and Ladders game.

The `while` loop's condition is `while square != finalSquare`, to reflect that you must land exactly on square 25.

```
gameLoop: while square != finalSquare {
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        // diceRoll will move us to the final square, so the game is over
        break gameLoop
    case let newSquare where newSquare > finalSquare:
        // diceRoll will move us beyond the final square, so roll again
        continue gameLoop
    default:
        // this is a valid move, so find out its effect
        square += diceRoll
        square += board[square]
    }
}
print("Game over!")
```

The dice is rolled at the start of each loop. Rather than moving the player immediately, the loop uses a `switch` statement to consider the result of the move and to determine whether the move is allowed:

- If the dice roll will move the player onto the final square, the game is over. The `break gameLoop` statement transfers control to the first line of code outside of the `while` loop, which ends the game.
- If the dice roll will move the player *beyond* the final square, the move is invalid and the player needs to roll again. The `continue gameLoop` statement ends the current `while` loop iteration and begins the next iteration of the loop.
- In all other cases, the dice roll is a valid move. The player moves forward by `diceRoll` squares, and the game logic checks for any snakes and ladders. The

loop then ends, and control returns to the `while` condition to decide whether another turn is required.

> NOTE
>
> If the `break` statement above didn't use the `gameLoop` label, it would break out of the `switch` statement, not the `while` statement. Using the `gameLoop` label makes it clear which control statement should be terminated.
>
> It isn't strictly necessary to use the `gameLoop` label when calling `continue gameLoop` to jump to the next iteration of the loop. there's only one loop in the game, and therefore no ambiguity as to which loop the `continue` statement will affect. However, there's no harm in using the `gameLoop` label with the `continue` statement. Doing so is consistent with the label's use alongside the `break` statement and helps make the game's logic clearer to read and understand.

Early Exit

A `guard` statement, like an `if` statement, executes statements depending on the Boolean value of an expression. You use a `guard` statement to require that a condition must be true in order for the code after the `guard` statement to be executed. Unlike an `if` statement, a `guard` statement always has an `else` clause—the code inside the `else` clause is executed if the condition isn't true.

```swift
func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }

    print("Hello \(name)!")

    guard let location = person["location"] else {
        print("I hope the weather is nice near you.")
        return
    }

    print("I hope the weather is nice in \(location).")
}

greet(person: ["name": "John"])
// Prints "Hello John!"
// Prints "I hope the weather is nice near you."
greet(person: ["name": "Jane", "location": "Cupertino"])
// Prints "Hello Jane!"
// Prints "I hope the weather is nice in Cupertino."
```

If the `guard` statement's condition is met, code execution continues after the `guard` statement's closing brace. Any variables or constants that were assigned values using an optional binding as part of the condition are available for the rest of the code block that the `guard` statement appears in.

If that condition isn't met, the code inside the `else` branch is executed. That branch must transfer control to exit the code block in which the `guard` statement appears. It can do this with a control transfer statement such as `return`, `break`, `continue`, or `throw`, or it can call a function or method that doesn't return, such as `fatalError(_:file:line:)`.

Using a `guard` statement for requirements improves the readability of your code, compared to doing the same check with an `if` statement. It lets you write the code that's typically executed without wrapping it in an `else` block, and it lets you keep the code that handles a violated requirement next to the requirement.

Checking API Availability

Swift has built-in support for checking API availability, which ensures that you don't accidentally use APIs that are unavailable on a given deployment target.

The compiler uses availability information in the SDK to verify that all of the APIs used in your code are available on the deployment target specified by your project. Swift reports an error at compile time if you try to use an API that isn't available.

You use an *availability condition* in an `if` or `guard` statement to conditionally execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

```
if #available(iOS 10, macOS 10.12, *) {
    // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS
} else {
    // Fall back to earlier iOS and macOS APIs
}
```

The availability condition above specifies that in iOS, the body of the `if` statement executes only in iOS 10 and later; in macOS, only in macOS 10.12 and later. The last argument, `*`, is required and specifies that on any other platform, the body of the `if` executes on the minimum deployment target specified by your target.

In its general form, the availability condition takes a list of platform names and versions. You use platform names such as `iOS`, `macOS`, `watchOS`, and `tvOS`—for the full list, see Declaration Attributes. In addition to specifying major version numbers like iOS 8 or macOS 10.10, you can specify minor versions numbers like iOS 11.2.6 and macOS 10.13.3.

```
if #available( platform name  version , ... , *) {
    statements to execute if the APIs are available
} else {

      fallback statements to execute if the APIs are unavailable

}
```

Functions

Functions are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to "call" the function to perform its task when needed.

Swift's unified function syntax is flexible enough to express anything from a simple C-style function with no parameter names to a complex Objective-C-style method with names and argument labels for each parameter. Parameters can provide default values to simplify function calls and can be passed as in-out parameters, which modify a passed variable once the function has completed its execution.

Every function in Swift has a type, consisting of the function's parameter types and return type. You can use this type like any other type in Swift, which makes it easy to pass functions as parameters to other functions, and to return functions from functions. Functions can also be written within other functions to encapsulate useful functionality within a nested function scope.

Defining and Calling Functions

When you define a function, you can optionally define one or more named, typed values that the function takes as input, known as *parameters*. You can also optionally define a type of value that the function will pass back as output when it's done, known as its *return type*.

Every function has a *function name*, which describes the task that the function performs. To use a function, you "call" that function with its name and pass it input values (known as *arguments*) that match the types of the function's parameters. A function's arguments must always be provided in the same order as the function's parameter list.

The function in the example below is called `greet(person:)`, because that's what it does—it takes a person's name as input and returns a greeting for that person. To accomplish this, you define one input parameter—a `String` value called `person`—and a return type of `String`, which will contain a greeting for that person:

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

All of this information is rolled up into the function's *definition*, which is prefixed with the `func` keyword. You indicate the function's return type with the *return arrow* `->` (a hyphen followed by a right angle bracket), which is followed by the name of the type to return.

The definition describes what the function does, what it expects to receive, and what it returns when it's done. The definition makes it easy for the function to be called unambiguously from elsewhere in your code:

```
print(greet(person: "Anna"))
// Prints "Hello, Anna!"
print(greet(person: "Brian"))
// Prints "Hello, Brian!"
```

You call the `greet(person:)` function by passing it a `String` value after the `person` argument label, such as `greet(person: "Anna")`. Because the function returns a `String` value, `greet(person:)` can be wrapped in a call to the `print(_:separator:terminator:)` function to print that string and see its return value, as shown above.

> NOTE
>
> The `print(_:separator:terminator:)` function doesn't have a label for its first argument, and its other arguments are optional because they have a default value. These variations on function syntax are discussed below in Function Argument Labels and Parameter Names and Default Parameter Values.

The body of the `greet(person:)` function starts by defining a new `String` constant called `greeting` and setting it to a simple greeting message. This greeting is then passed back out of the function using the `return` keyword. In the line of code that says `return greeting`, the function finishes its execution and returns the current value of `greeting`.

You can call the `greet(person:)` function multiple times with different input values. The example above shows what happens if it's called with an input value of `"Anna"`, and an input value of `"Brian"`. The function returns a tailored greeting in each case.

To make the body of this function shorter, you can combine the message creation and the return statement into one line:

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
// Prints "Hello again, Anna!"
```

Function Parameters and Return Values

Function parameters and return values are extremely flexible in Swift. You can define anything from a simple utility function with a single unnamed

parameter to a complex function with expressive parameter names and different parameter options.

Functions Without Parameters

Functions aren't required to define input parameters. Here's a function with no input parameters, which always returns the same `String` message whenever it's called:

```
1  func sayHelloWorld() -> String {
2      return "hello, world"
3  }
4  print(sayHelloWorld())
5  // Prints "hello, world"
```

The function definition still needs parentheses after the function's name, even though it doesn't take any parameters. The function name is also followed by an empty pair of parentheses when the function is called.

Functions With Multiple Parameters

Functions can have multiple input parameters, which are written within the function's parentheses, separated by commas.

This function takes a person's name and whether they have already been greeted as input, and returns an appropriate greeting for that person:

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Prints "Hello again, Tim!"
```

You call the `greet(person:alreadyGreeted:)` function by passing it both a `String` argument value labeled `person` and a `Bool` argument value labeled `alreadyGreeted` in parentheses, separated by commas. Note that this function is distinct from the `greet(person:)` function shown in an earlier section. Although both functions have names that begin with `greet`, the `greet(person:alreadyGreeted:)` function takes two arguments but the `greet(person:)` function takes only one.

Functions Without Return Values

Functions aren't required to define a return type. Here's a version of the `greet(person:)` function, which prints its own `String` value rather than returning it:

```
func greet(person: String) {
    print("Hello, \(person)!")
}
greet(person: "Dave")
// Prints "Hello, Dave!"
```

Because it doesn't need to return a value, the function's definition doesn't include the return arrow (`->`) or a return type.

> NOTE
>
> Strictly speaking, this version of the `greet(person:)` function *does* still return a value, even though no return value is defined. Functions without a defined return type return a special value of type `Void`. This is simply an empty tuple, which is written as `()`.

The return value of a function can be ignored when it's called:

```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting(string: "hello, world")
// prints "hello, world" but doesn't return a value
```

The first function, `printAndCount(string:)`, prints a string, and then returns its character count as an `Int`. The second function, `printWithoutCounting(string:)`, calls the first function, but ignores its return value. When the second function is called, the message is still printed by the first function, but the returned value isn't used.

> NOTE
>
> Return values can be ignored, but a function that says it will return a value must always do so. A function with a defined return type can't allow control to fall out of the bottom of the function without returning a value, and attempting to do so will result in a compile-time error.

Functions with Multiple Return Values

You can use a tuple type as the return type for a function to return multiple values as part of one compound return value.

The example below defines a function called `minMax(array:)`, which finds the smallest and largest numbers in an array of `Int` values:

```
1   func minMax(array: [Int]) -> (min: Int, max: Int) {
2       var currentMin = array[0]
3       var currentMax = array[0]
4       for value in array[1..<array.count] {
5           if value < currentMin {
6               currentMin = value
7           } else if value > currentMax {
8               currentMax = value
9           }
10      }
11      return (currentMin, currentMax)
12  }
```

The `minMax(array:)` function returns a tuple containing two `Int` values. These values are labeled `min` and `max` so that they can be accessed by name when querying the function's return value.

The body of the `minMax(array:)` function starts by setting two working variables called `currentMin` and `currentMax` to the value of the first integer in the array. The function then iterates over the remaining values in the array and checks each value to see if it's smaller or larger than the values of `currentMin` and `currentMax` respectively. Finally, the overall minimum and maximum values are returned as a tuple of two `Int` values.

Because the tuple's member values are named as part of the function's return type, they can be accessed with dot syntax to retrieve the minimum and maximum found values:

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \
      (bounds.max)")
// Prints "min is -6 and max is 109"
```

Note that the tuple's members don't need to be named at the point that the tuple is returned from the function, because their names are already specified as part of the function's return type.

Optional Tuple Return Types

If the tuple type to be returned from a function has the potential to have "no value" for the entire tuple, you can use an *optional* tuple return type to reflect the fact that the entire tuple can be `nil`. You write an optional tuple return type by placing a question mark after the tuple type's closing parenthesis, such as `(Int, Int)?` or `(String, Int, Bool)?`.

> NOTE
>
> An optional tuple type such as `(Int, Int)?` is different from a tuple that contains optional types such as `(Int?, Int?)`. With an optional tuple type, the entire tuple is optional, not just each individual value within the tuple.

The `minMax(array:)` function above returns a tuple containing two `Int` values. However, the function doesn't perform any safety checks on the array it's passed. If the `array` argument contains an empty array, the `minMax(array:)` function, as defined above, will trigger a runtime error when attempting to access `array[0]`.

To handle an empty array safely, write the `minMax(array:)` function with an optional tuple return type and return a value of `nil` when the array is empty:

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

You can use optional binding to check whether this version of the `minMax(array:)` function returns an actual tuple value or `nil`:

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
// Prints "min is -6 and max is 109"
```

Functions With an Implicit Return

If the entire body of the function is a single expression, the function implicitly returns that expression. For example, both functions below have

the same behavior:

```
1  func greeting(for person: String) -> String {
2      "Hello, " + person + "!"
3  }
4  print(greeting(for: "Dave"))
5  // Prints "Hello, Dave!"
6
7  func anotherGreeting(for person: String) -> String {
8      return "Hello, " + person + "!"
9  }
10 print(anotherGreeting(for: "Dave"))
11 // Prints "Hello, Dave!"
```

The entire definition of the `greeting(for:)` function is the greeting message that it returns, which means it can use this shorter form. The `anotherGreeting(for:)` function returns the same greeting message, using the `return` keyword like a longer function. Any function that you write as just one `return` line can omit the `return`.

As you'll see in Shorthand Getter Declaration, property getters can also use an implicit return.

> NOTE
>
> The code you write as an implicit return value needs to return some value. For example, you can't use `fatalError("Oh no!")` or `print(13)` as an implicit return value.

Function Argument Labels and Parameter Names

Each function parameter has both an *argument label* and a *parameter name*. The argument label is used when calling the function; each argument is written in the function call with its argument label before it. The parameter name is used in the implementation of the function. By default, parameters use their parameter name as their argument label.

```
1  func someFunction(firstParameterName: Int,
       secondParameterName: Int) {
2      // In the function body, firstParameterName and
       secondParameterName
3      // refer to the argument values for the first
       and second parameters.
4  }
5  someFunction(firstParameterName: 1,
       secondParameterName: 2)
```

All parameters must have unique names. Although it's possible for multiple parameters to have the same argument label, unique argument labels help make your code more readable.

Specifying Argument Labels

You write an argument label before the parameter name, separated by a space:

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}
```

Here's a variation of the `greet(person:)` function that takes a person's name and hometown and returns a greeting:

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino."
```

The use of argument labels can allow a function to be called in an expressive, sentence-like manner, while still providing a function body that's readable and clear in intent.

Omitting Argument Labels

If you don't want an argument label for a parameter, write an underscore (_) instead of an explicit argument label for that parameter.

```
1  func someFunction(_ firstParameterName: Int,
       secondParameterName: Int) {
2      // In the function body, firstParameterName and
       secondParameterName
3      // refer to the argument values for the first
       and second parameters.
4  }
5  someFunction(1, secondParameterName: 2)
```

If a parameter has an argument label, the argument *must* be labeled when you call the function.

Default Parameter Values

You can define a *default value* for any parameter in a function by assigning a value to the parameter after that parameter's type. If a default value is defined, you can omit that parameter when calling the function.

```
1  func someFunction(parameterWithoutDefault: Int,
       parameterWithDefault: Int = 12) {
2      // If you omit the second argument when calling
       this function, then
3      // the value of parameterWithDefault is 12
       inside the function body.
4  }
5  someFunction(parameterWithoutDefault: 3,
       parameterWithDefault: 6) //
       parameterWithDefault is 6
6  someFunction(parameterWithoutDefault: 4) //
       parameterWithDefault is 12
```

Place parameters that don't have default values at the beginning of a function's parameter list, before the parameters that have default values. Parameters that don't have default values are usually more important to the function's meaning—writing them first makes it easier to recognize that the same function is being called, regardless of whether any default parameters are omitted.

Variadic Parameters

A *variadic parameter* accepts zero or more values of a specified type. You use a variadic parameter to specify that the parameter can be passed a varying number of input values when the function is called. Write variadic parameters by inserting three period characters (`...`) after the parameter's type name.

The values passed to a variadic parameter are made available within the function's body as an array of the appropriate type. For example, a variadic parameter with a name of `numbers` and a type of `Double...` is made

available within the function's body as a constant array called `numbers` of type `[Double]`.

The example below calculates the *arithmetic mean* (also known as the *average*) for a list of numbers of any length:

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

A function can have multiple variadic parameters. The first parameter that comes after a variadic parameter must have an argument label. The argument label makes it unambiguous which arguments are passed to the variadic parameter and which arguments are passed to the parameters that come after the variadic parameter.

In-Out Parameters

Function parameters are constants by default. Trying to change the value of a function parameter from within the body of that function results in a compile-time error. This means that you can't change the value of a parameter by mistake. If you want a function to modify a parameter's value, and you want those changes to persist after the function call has ended, define that parameter as an *in-out parameter* instead.

You write an in-out parameter by placing the `inout` keyword right before a parameter's type. An in-out parameter has a value that's passed *in* to the function, is modified by the function, and is passed back *out* of the function to replace the original value. For a detailed discussion of the behavior of in-out parameters and associated compiler optimizations, see [In-Out Parameters].

You can only pass a variable as the argument for an in-out parameter. You can't pass a constant or a literal value as the argument, because constants and literals can't be modified. You place an ampersand (`&`) directly before a variable's name when you pass it as an argument to an in-out parameter, to indicate that it can be modified by the function.

> NOTE
>
> In-out parameters can't have default values, and variadic parameters can't be marked as `inout`.

Here's an example of a function called `swapTwoInts(_:_:)`, which has two in-out integer parameters called `a` and `b`:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

The `swapTwoInts(_:_:)` function simply swaps the value of `b` into `a`, and the value of `a` into `b`. The function performs this swap by storing the value of `a` in a temporary constant called `temporaryA`, assigning the value of `b` to `a`, and then assigning `temporaryA` to `b`.

You can call the `swapTwoInts(_:_:)` function with two variables of type `Int` to swap their values. Note that the names of `someInt` and `anotherInt` are prefixed with an ampersand when they're passed to the `swapTwoInts(_:_:)` function:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

The example above shows that the original values of `someInt` and `anotherInt` are modified by the `swapTwoInts(_:_:)` function, even though they were originally defined outside of the function.

> NOTE
>
> In-out parameters aren't the same as returning a value from a function. The `swapTwoInts` example above doesn't define a return type or return a value, but it still modifies the values of `someInt` and `anotherInt`. In-out parameters are an alternative way for a function to have an effect outside of the scope of its function body.

Function Types

Every function has a specific *function type*, made up of the parameter types and the return type of the function.

For example:

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

This example defines two simple mathematical functions called `addTwoInts` and `multiplyTwoInts`. These functions each take two `Int` values, and return an `Int` value, which is the result of performing an appropriate mathematical operation.

The type of both of these functions is `(Int, Int) -> Int`. This can be read as:

"A function that has two parameters, both of type `Int`, and that returns a value of type `Int`."

Here's another example, for a function with no parameters or return value:

```
func printHelloWorld() {
    print("hello, world")
}
```

The type of this function is `() -> Void`, or "a function that has no parameters, and returns `Void`."

Using Function Types

You use function types just like any other types in Swift. For example, you can define a constant or variable to be of a function type and assign an appropriate function to that variable:

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

This can be read as:

"Define a variable called `mathFunction`, which has a type of 'a function that takes two `Int` values, and returns an `Int` value.' Set this new variable to refer to the function called `addTwoInts`."

The `addTwoInts(_:_:)` function has the same type as the `mathFunction` variable, and so this assignment is allowed by Swift's type-checker.

You can now call the assigned function with the name `mathFunction`:

```
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"
```

A different function with the same matching type can be assigned to the same variable, in the same way as for nonfunction types:

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"
```

As with any other type, you can leave it to Swift to infer the function type when you assign a function to a constant or variable:

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type
     (Int, Int) -> Int
```

Function Types as Parameter Types

You can use a function type such as `(Int, Int) -> Int` as a parameter type for another function. This enables you to leave some aspects of a function's implementation for the function's caller to provide when the function is called.

Here's an example to print the results of the math functions from above:

```
func printMathResult(_ mathFunction: (Int, Int) ->
     Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Prints "Result: 8"
```

This example defines a function called `printMathResult(_:_:_:)`, which has three parameters. The first parameter is called `mathFunction`, and is of type `(Int, Int) -> Int`. You can pass any function of that type as the argument for this first parameter. The second and third parameters are called `a` and `b`, and are both of type `Int`. These are used as the two input values for the provided math function.

When `printMathResult(_:_:_:)` is called, it's passed the `addTwoInts(_:_:)` function, and the integer values `3` and `5`. It calls the provided function with the values `3` and `5`, and prints the result of `8`.

The role of `printMathResult(_:_:_:)` is to print the result of a call to a math function of an appropriate type. It doesn't matter what that function's implementation actually does—it matters only that the function is of the correct type. This enables `printMathResult(_:_:_:)` to hand off some of its functionality to the caller of the function in a type-safe way.

Function Types as Return Types

You can use a function type as the return type of another function. You do this by writing a complete function type immediately after the return arrow (`->`) of the returning function.

The next example defines two simple functions called `stepForward(_:)` and `stepBackward(_:)`. The `stepForward(_:)` function returns a value one more than its input value, and the `stepBackward(_:)` function returns a value one less than its input value. Both functions have a type of `(Int) -> Int`:

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

Here's a function called `chooseStepFunction(backward:)`, whose return type is `(Int) -> Int`. The `chooseStepFunction(backward:)` function returns the `stepForward(_:)` function or the `stepBackward(_:)` function based on a Boolean parameter called `backward`:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

You can now use `chooseStepFunction(backward:)` to obtain a function that will step in one direction or the other:

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

The example above determines whether a positive or negative step is needed to move a variable called `currentValue` progressively closer to zero. `currentValue` has an initial value of `3`, which means that `currentValue > 0` returns `true`, causing `chooseStepFunction(backward:)` to return the `stepBackward(_:)` function. A reference to the returned function is stored in a constant called `moveNearerToZero`.

Now that `moveNearerToZero` refers to the correct function, it can be used to count to zero:

```
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// 3...
// 2...
// 1...
// zero!
```

Nested Functions

All of the functions you have encountered so far in this chapter have been examples of *global functions*, which are defined at a global scope. You can also define functions inside the bodies of other functions, known as *nested functions*.

Nested functions are hidden from the outside world by default, but can still be called and used by their enclosing function. An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope.

You can rewrite the `chooseStepFunction(backward:)` example above to use and return nested functions:

```
1  func chooseStepFunction(backward: Bool) -> (Int) ->
       Int {
2      func stepForward(input: Int) -> Int { return
       input + 1 }
3      func stepBackward(input: Int) -> Int { return
       input - 1 }
4      return backward ? stepBackward : stepForward
5  }
6  var currentValue = -4
7  let moveNearerToZero = chooseStepFunction(backward:
       currentValue > 0)
8  // moveNearerToZero now refers to the nested
       stepForward() function
9  while currentValue != 0 {
10     print("\(currentValue)... ")
11     currentValue = moveNearerToZero(currentValue)
12 }
13 print("zero!")
14 // -4...
15 // -3...
16 // -2...
17 // -1...
18 // zero!
```

Closures

Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.

Closures can capture and store references to any constants and variables from the context in which they're defined. This is known as *closing over* those constants and variables. Swift handles all of the memory management of capturing for you.

> NOTE
>
> Don't worry if you aren't familiar with the concept of capturing. It's explained in detail below in Capturing Values.

Global and nested functions, as introduced in Functions, are actually special cases of closures. Closures take one of three forms:

- Global functions are closures that have a name and don't capture any values.
- Nested functions are closures that have a name and can capture values from their enclosing function.
- Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

Swift's closure expressions have a clean, clear style, with optimizations that encourage brief, clutter-free syntax in common scenarios. These optimizations include:

- Inferring parameter and return value types from context
- Implicit returns from single-expression closures

- Shorthand argument names
- Trailing closure syntax

Closure Expressions

Nested functions, as introduced in Nested Functions, are a convenient means of naming and defining self-contained blocks of code as part of a larger function. However, it's sometimes useful to write shorter versions of function-like constructs without a full declaration and name. This is particularly true when you work with functions or methods that take functions as one or more of their arguments.

Closure expressions are a way to write inline closures in a brief, focused syntax. Closure expressions provide several syntax optimizations for writing closures in a shortened form without loss of clarity or intent. The closure expression examples below illustrate these optimizations by refining a single example of the `sorted(by:)` method over several iterations, each of which expresses the same functionality in a more succinct way.

The Sorted Method

Swift's standard library provides a method called `sorted(by:)`, which sorts an array of values of a known type, based on the output of a sorting closure that you provide. Once it completes the sorting process, the `sorted(by:)` method returns a new array of the same type and size as the old one, with its elements in the correct sorted order. The original array isn't modified by the `sorted(by:)` method.

The closure expression examples below use the `sorted(by:)` method to sort an array of `String` values in reverse alphabetical order. Here's the initial array to be sorted:

```
let names = ["Chris", "Alex", "Ewa", "Barry",
     "Daniella"]
```

The `sorted(by:)` method accepts a closure that takes two arguments of the same type as the array's contents, and returns a `Bool` value to say whether the first value should appear before or after the second value once the values are sorted. The sorting closure needs to return `true` if the first value should appear *before* the second value, and `false` otherwise.

This example is sorting an array of `String` values, and so the sorting closure needs to be a function of type `(String, String) -> Bool`.

One way to provide the sorting closure is to write a normal function of the correct type, and to pass it in as an argument to the `sorted(by:)` method:

```
1  func backward(_ s1: String, _ s2: String) -> Bool {
2      return s1 > s2
3  }
4  var reversedNames = names.sorted(by: backward)
5  // reversedNames is equal to ["Ewa", "Daniella",
       "Chris", "Barry", "Alex"]
```

If the first string (`s1`) is greater than the second string (`s2`), the `backward(_:_:)` function will return `true`, indicating that `s1` should appear before `s2` in the sorted array. For characters in strings, "greater than" means "appears later in the alphabet than". This means that the letter `"B"` is "greater than" the letter `"A"`, and the string `"Tom"` is greater than the string `"Tim"`. This gives a reverse alphabetical sort, with `"Barry"` being placed before `"Alex"`, and so on.

However, this is a rather long-winded way to write what is essentially a single-expression function (`a > b`). In this example, it would be preferable to write the sorting closure inline, using closure expression syntax.

Closure Expression Syntax
Closure expression syntax has the following general form:

```
{ (parameters) -> return type in
    statements
}
```

The *parameters* in closure expression syntax can be in-out parameters, but they can't have a default value. Variadic parameters can be used if you name the variadic parameter. Tuples can also be used as parameter types and return types.

The example below shows a closure expression version of the `backward(_:_:)` function from above:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})
```

Note that the declaration of parameters and return type for this inline closure is identical to the declaration from the `backward(_:_:)` function. In both cases, it's written as `(s1: String, s2: String) -> Bool`. However, for the inline closure expression, the parameters and return type are written *inside* the curly braces, not outside of them.

The start of the closure's body is introduced by the `in` keyword. This keyword indicates that the definition of the closure's parameters and return type has finished, and the body of the closure is about to begin.

Because the body of the closure is so short, it can even be written on a single line:

```
reversedNames = names.sorted(by: { (s1: String, s2:
     String) -> Bool in return s1 > s2 } )
```

This illustrates that the overall call to the `sorted(by:)` method has remained the same. A pair of parentheses still wrap the entire argument for the method. However, that argument is now an inline closure.

Inferring Type From Context

Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns. The `sorted(by:)` method is being called on an array of strings, so its argument must be a function of type `(String, String) -> Bool`. This means that the `(String, String)` and `Bool` types don't need to be written as part of the closure expression's definition. Because all of the types can be inferred, the return arrow (`->`) and the parentheses around the names of the parameters can also be omitted:

```
reversedNames = names.sorted(by: { s1, s2 in return
     s1 > s2 } )
```

It's always possible to infer the parameter types and return type when passing a closure to a function or method as an inline closure expression. As a result, you never need to write an inline closure in its fullest form when the closure is used as a function or method argument.

Nonetheless, you can still make the types explicit if you wish, and doing so is encouraged if it avoids ambiguity for readers of your code. In the case of the `sorted(by:)` method, the purpose of the closure is clear from the fact that sorting is taking place, and it's safe for a reader to assume that the closure is likely to be working with `String` values, because it's assisting with the sorting of an array of strings.

Implicit Returns from Single-Expression Closures
Single-expression closures can implicitly return the result of their single expression by omitting the `return` keyword from their declaration, as in this version of the previous example:

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2
     } )
```

Here, the function type of the `sorted(by:)` method's argument makes it clear that a `Bool` value must be returned by the closure. Because the closure's body contains a single expression (`s1 > s2`) that returns a `Bool` value, there's no ambiguity, and the `return` keyword can be omitted.

Shorthand Argument Names
Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names `$0`, `$1`, `$2`, and so on.

If you use these shorthand argument names within your closure expression, you can omit the closure's argument list from its definition. The type of the shorthand argument names is inferred from the expected function type, and the highest numbered shorthand argument you use determines the number of arguments that the closure takes. The `in` keyword can also be omitted, because the closure expression is made up entirely of its body:

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

Here, `$0` and `$1` refer to the closure's first and second `String` arguments. Because `$1` is the shorthand argument with highest number, the closure is understood to take two arguments. Because the `sorted(by:)` function here expects a closure whose arguments are both strings, the shorthand arguments `$0` and `$1` are both of type `String`.

Operator Methods
There's actually an even *shorter* way to write the closure expression above. Swift's `String` type defines its string-specific implementation of the greater-than operator (`>`) as a method that has two parameters of type `String`, and returns a value of type `Bool`. This exactly matches the method type needed by the `sorted(by:)` method. Therefore, you can simply pass in the greater-than operator, and Swift will infer that you want to use its string-specific implementation:

```
reversedNames = names.sorted(by: >)
```

For more about operator methods, see Operator Methods.

Trailing Closures

If you need to pass a closure expression to a function as the function's final argument and the closure expression is long, it can be useful to write it as a *trailing closure* instead. You write a trailing closure after the function call's parentheses, even though the trailing closure is still an argument to the function. When you use the trailing closure syntax, you don't write the argument label for the first closure as part of the function call. A function call can include multiple trailing closures; however, the first few examples below use a single trailing closure.

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

The string-sorting closure from the Closure Expression Syntax section above can be written outside of the `sorted(by:)` method's parentheses as a trailing closure:

```
reversedNames = names.sorted() { $0 > $1 }
```

If a closure expression is provided as the function's or method's only argument and you provide that expression as a trailing closure, you don't

need to write a pair of parentheses `()` after the function or method's name when you call the function:

```
reversedNames = names.sorted { $0 > $1 }
```

Trailing closures are most useful when the closure is sufficiently long that it isn't possible to write it inline on a single line. As an example, Swift's `Array` type has a `map(_:)` method, which takes a closure expression as its single argument. The closure is called once for each item in the array, and returns an alternative mapped value (possibly of some other type) for that item. You specify the nature of the mapping and the type of the returned value by writing code in the closure that you pass to `map(_:)`.

After applying the provided closure to each array element, the `map(_:)` method returns a new array containing all of the new mapped values, in the same order as their corresponding values in the original array.

Here's how you can use the `map(_:)` method with a trailing closure to convert an array of `Int` values into an array of `String` values. The array `[16, 58, 510]` is used to create the new array `["OneSix", "FiveEight", "FiveOneZero"]`:

```
let digitNames = [
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
let numbers = [16, 58, 510]
```

The code above creates a dictionary of mappings between the integer digits and English-language versions of their names. It also defines an array of integers, ready to be converted into strings.

You can now use the `numbers` array to create an array of `String` values, by passing a closure expression to the array's `map(_:)` method as a trailing closure:

```
let strings = numbers.map { (number) -> String in
    var number = number
    var output = ""
    repeat {
        output = digitNames[number % 10]! + output
        number /= 10
    } while number > 0
    return output
}
// strings is inferred to be of type [String]
// its value is ["OneSix", "FiveEight",
     "FiveOneZero"]
```

The `map(_:)` method calls the closure expression once for each item in the array. You don't need to specify the type of the closure's input parameter, `number`, because the type can be inferred from the values in the array to be mapped.

In this example, the variable `number` is initialized with the value of the closure's `number` parameter, so that the value can be modified within the closure body. (The parameters to functions and closures are always constants.) The closure expression also specifies a return type of `String`, to indicate the type that will be stored in the mapped output array.

The closure expression builds a string called `output` each time it's called. It calculates the last digit of `number` by using the remainder operator (`number % 10`), and uses this digit to look up an appropriate string in the

`digitNames` dictionary. The closure can be used to create a string representation of any integer greater than zero.

> NOTE
>
> The call to the `digitNames` dictionary's subscript is followed by an exclamation point (`!`), because dictionary subscripts return an optional value to indicate that the dictionary lookup can fail if the key doesn't exist. In the example above, it's guaranteed that `number % 10` will always be a valid subscript key for the `digitNames` dictionary, and so an exclamation point is used to force-unwrap the `String` value stored in the subscript's optional return value.

The string retrieved from the `digitNames` dictionary is added to the *front* of `output`, effectively building a string version of the number in reverse. (The expression `number % 10` gives a value of `6` for `16`, `8` for `58`, and `0` for `510`.)

The `number` variable is then divided by `10`. Because it's an integer, it's rounded down during the division, so `16` becomes `1`, `58` becomes `5`, and `510` becomes `51`.

The process is repeated until `number` is equal to `0`, at which point the `output` string is returned by the closure, and is added to the output array by the `map(_:)` method.

The use of trailing closure syntax in the example above neatly encapsulates the closure's functionality immediately after the function that closure supports, without needing to wrap the entire closure within the `map(_:)` method's outer parentheses.

If a function takes multiple closures, you omit the argument label for the first trailing closure and you label the remaining trailing closures. For example, the function below loads a picture for a photo gallery:

```
func loadPicture(from server: Server, completion: (Picture) -> Void, onFailure: () -> Void) {
    if let picture = download("photo.jpg", from: server) {
        completion(picture)
    } else {
        onFailure()
    }
}
```

When you call this function to load a picture, you provide two closures. The first closure is a completion handler that displays a picture after a successful download. The second closure is an error handler that displays an error to the user.

```
loadPicture(from: someServer) { picture in
    someView.currentPicture = picture
} onFailure: {
    print("Couldn't download the next picture.")
}
```

In this example, the `loadPicture(from:completion:onFailure:)` function dispatches its network task into the background, and calls one of the two completion handlers when the network task finishes. Writing the function this way lets you cleanly separate the code that's responsible for handling a network failure from the code that updates the user interface after a successful download, instead of using just one closure that handles both circumstances.

Capturing Values

A closure can *capture* constants and variables from the surrounding context in which it's defined. The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists.

In Swift, the simplest form of a closure that can capture values is a nested function, written within the body of another function. A nested function can capture any of its outer function's arguments and can also capture any constants and variables defined within the outer function.

Here's an example of a function called `makeIncrementer`, which contains a nested function called `incrementer`. The nested `incrementer()` function captures two values, `runningTotal` and `amount`, from its surrounding context. After capturing these values, `incrementer` is returned by `makeIncrementer` as a closure that increments `runningTotal` by `amount` each time it's called.

```
1  func makeIncrementer(forIncrement amount: Int) -> ()
       -> Int {
2      var runningTotal = 0
3      func incrementer() -> Int {
4          runningTotal += amount
5          return runningTotal
6      }
7      return incrementer
8  }
```

The return type of `makeIncrementer` is `() -> Int`. This means that it returns a *function*, rather than a simple value. The function it returns has no

parameters, and returns an `Int` value each time it's called. To learn how functions can return other functions, see [Function Types as Return Types](#).

The `makeIncrementer(forIncrement:)` function defines an integer variable called `runningTotal`, to store the current running total of the incrementer that will be returned. This variable is initialized with a value of `0`.

The `makeIncrementer(forIncrement:)` function has a single `Int` parameter with an argument label of `forIncrement`, and a parameter name of `amount`. The argument value passed to this parameter specifies how much `runningTotal` should be incremented by each time the returned incrementer function is called. The `makeIncrementer` function defines a nested function called `incrementer`, which performs the actual incrementing. This function simply adds `amount` to `runningTotal`, and returns the result.

When considered in isolation, the nested `incrementer()` function might seem unusual:

```
1  func incrementer() -> Int {
2      runningTotal += amount
3      return runningTotal
4  }
```

The `incrementer()` function doesn't have any parameters, and yet it refers to `runningTotal` and `amount` from within its function body. It does this by capturing a *reference* to `runningTotal` and `amount` from the surrounding function and using them within its own function body. Capturing by reference ensures that `runningTotal` and `amount` don't disappear when the call to `makeIncrementer` ends, and also ensures that `runningTotal` is available the next time the `incrementer` function is called.

> NOTE
>
> As an optimization, Swift may instead capture and store a *copy* of a value if that value isn't mutated by a closure, and if the value isn't mutated after the closure is created.
>
> Swift also handles all memory management involved in disposing of variables when they're no longer needed.

Here's an example of `makeIncrementer` in action:

```
let incrementByTen = makeIncrementer(forIncrement:
     10)
```

This example sets a constant called `incrementByTen` to refer to an incrementer function that adds `10` to its `runningTotal` variable each time it's called. Calling the function multiple times shows this behavior in action:

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

If you create a second incrementer, it will have its own stored reference to a new, separate `runningTotal` variable:

```
let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()
// returns a value of 7
```

Calling the original incrementer (`incrementByTen`) again continues to increment its own `runningTotal` variable, and doesn't affect the variable captured by `incrementBySeven`:

```
incrementByTen()
// returns a value of 40
```

> NOTE
>
> If you assign a closure to a property of a class instance, and the closure captures that instance by referring to the instance or its members, you will create a strong reference cycle between the closure and the instance. Swift uses *capture lists* to break these strong reference cycles. For more information, see Strong Reference Cycles for Closures.

Closures Are Reference Types

In the example above, `incrementBySeven` and `incrementByTen` are constants, but the closures these constants refer to are still able to increment the `runningTotal` variables that they have captured. This is because functions and closures are *reference types*.

Whenever you assign a function or a closure to a constant or a variable, you are actually setting that constant or variable to be a *reference* to the function or closure. In the example above, it's the choice of closure that

`incrementByTen` *refers to* that's constant, and not the contents of the closure itself.

This also means that if you assign a closure to two different constants or variables, both of those constants or variables refer to the same closure.

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns a value of 50

incrementByTen()
// returns a value of 60
```

The example above shows that calling `alsoIncrementByTen` is the same as calling `incrementByTen`. Because both of them refer to the same closure, they both increment and return the same running total.

Escaping Closures

A closure is said to *escape* a function when the closure is passed as an argument to the function, but is called after the function returns. When you declare a function that takes a closure as one of its parameters, you can write `@escaping` before the parameter's type to indicate that the closure is allowed to escape.

One way that a closure can escape is by being stored in a variable that's defined outside the function. As an example, many functions that start an asynchronous operation take a closure argument as a completion handler. The function returns after it starts the operation, but the closure isn't called until the operation is completed—the closure needs to escape, to be called later. For example:

```
var completionHandlers: [() -> Void] = []
func
     someFunctionWithEscapingClosure(completionHandl
     er: @escaping () -> Void) {
    completionHandlers.append(completionHandler)
}
```

The `someFunctionWithEscapingClosure(_:)` function takes a closure as its argument and adds it to an array that's declared outside the function. If you didn't mark the parameter of this function with `@escaping`, you would get a compile-time error.

An escaping closure that refers to `self` needs special consideration if `self` refers to an instance of a class. Capturing `self` in an escaping closure makes it easy to accidentally create a strong reference cycle. For information about reference cycles, see Automatic Reference Counting.

Normally, a closure captures variables implicitly by using them in the body of the closure, but in this case you need to be explicit. If you want to capture `self`, write `self` explicitly when you use it, or include `self` in the closure's capture list. Writing `self` explicitly lets you express your intent, and reminds you to confirm that there isn't a reference cycle. For example, in the code below, the closure passed to `someFunctionWithEscapingClosure(_:)` refers to `self` explicitly. In contrast, the closure passed to `someFunctionWithNonescapingClosure(_:)` is a nonescaping closure, which means it can refer to `self` implicitly.

```
1  func someFunctionWithNonescapingClosure(closure: ()
       -> Void) {
2      closure()
3  }
4
5  class SomeClass {
6      var x = 10
7      func doSomething() {
8          someFunctionWithEscapingClosure { self.x =
       100 }
9          someFunctionWithNonescapingClosure { x = 200
       }
10     }
11 }
12
13 let instance = SomeClass()
14 instance.doSomething()
15 print(instance.x)
16 // Prints "200"
17
18 completionHandlers.first?()
19 print(instance.x)
20 // Prints "100"
```

Here's a version of `doSomething()` that captures `self` by including it in the closure's capture list, and then refers to `self` implicitly:

```
class SomeOtherClass {
    var x = 10
    func doSomething() {
        someFunctionWithEscapingClosure { [self] in x = 100 }
        someFunctionWithNonescapingClosure { x = 200 }
    }
}
```

If `self` is an instance of a structure or an enumeration, you can always refer to `self` implicitly. However, an escaping closure can't capture a mutable reference to `self` when `self` is an instance of a structure or an enumeration. Structures and enumerations don't allow shared mutability, as discussed in [Structures and Enumerations Are Value Types](#).

```
struct SomeStruct {
    var x = 10
    mutating func doSomething() {
        someFunctionWithNonescapingClosure { x = 200 }  // Ok
        someFunctionWithEscapingClosure { x = 100 }  // Error
    }
}
```

The call to the `someFunctionWithEscapingClosure` function in the example above is an error because it's inside a mutating method, so `self` is

mutable. That violates the rule that escaping closures can't capture a mutable reference to `self` for structures.

Autoclosures

An *autoclosure* is a closure that's automatically created to wrap an expression that's being passed as an argument to a function. It doesn't take any arguments, and when it's called, it returns the value of the expression that's wrapped inside of it. This syntactic convenience lets you omit braces around a function's parameter by writing a normal expression instead of an explicit closure.

It's common to *call* functions that take autoclosures, but it's not common to *implement* that kind of function. For example, the `assert(condition:message:file:line:)` function takes an autoclosure for its `condition` and `message` parameters; its `condition` parameter is evaluated only in debug builds and its `message` parameter is evaluated only if `condition` is `false`.

An autoclosure lets you delay evaluation, because the code inside isn't run until you call the closure. Delaying evaluation is useful for code that has side effects or is computationally expensive, because it lets you control when that code is evaluated. The code below shows how a closure delays evaluation.

```
var customersInLine = ["Chris", "Alex", "Ewa",
     "Barry", "Daniella"]
print(customersInLine.count)
// Prints "5"

let customerProvider = { customersInLine.remove(at:
     0) }
print(customersInLine.count)
// Prints "5"

print("Now serving \(customerProvider())!")
// Prints "Now serving Chris!"
print(customersInLine.count)
// Prints "4"
```

Even though the first element of the `customersInLine` array is removed by the code inside the closure, the array element isn't removed until the closure is actually called. If the closure is never called, the expression inside the closure is never evaluated, which means the array element is never removed. Note that the type of `customerProvider` isn't `String` but `() -> String`—a function with no parameters that returns a string.

You get the same behavior of delayed evaluation when you pass a closure as an argument to a function.

```
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: { customersInLine.remove(at: 0) } )
// Prints "Now serving Alex!"
```

The `serve(customer:)` function in the listing above takes an explicit closure that returns a customer's name. The version of `serve(customer:)` below performs the same operation but, instead of taking an explicit closure, it takes an autoclosure by marking its parameter's type with the `@autoclosure` attribute. Now you can call the function as if it took a `String` argument instead of a closure. The argument is automatically converted to a closure, because the `customerProvider` parameter's type is marked with the `@autoclosure` attribute.

```
// customersInLine is ["Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: @autoclosure () -> String) {
    print("Now serving \(customerProvider())!")
}
serve(customer: customersInLine.remove(at: 0))
// Prints "Now serving Ewa!"
```

> NOTE
>
> Overusing autoclosures can make your code hard to understand. The context and function name should make it clear that evaluation is being deferred.

If you want an autoclosure that's allowed to escape, use both the `@autoclosure` and `@escaping` attributes. The `@escaping` attribute is described above in [Escaping Closures](#).

```
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []
func collectCustomerProviders(_ customerProvider:
     @autoclosure @escaping () -> String) {
    customerProviders.append(customerProvider)
}
collectCustomerProviders(customersInLine.remove(at:
     0))
collectCustomerProviders(customersInLine.remove(at:
     0))

print("Collected \(customerProviders.count)
     closures.")
// Prints "Collected 2 closures."
for customerProvider in customerProviders {
    print("Now serving \(customerProvider())!")
}
// Prints "Now serving Barry!"
// Prints "Now serving Daniella!"
```

In the code above, instead of calling the closure passed to it as its `customerProvider` argument, the `collectCustomerProviders(_:)` function appends the closure to the `customerProviders` array. The array is declared outside the scope of the function, which means the closures in the

array can be executed after the function returns. As a result, the value of the `customerProvider` argument must be allowed to escape the function's scope.

Enumerations

An *enumeration* defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.

If you are familiar with C, you will know that C enumerations assign related names to a set of integer values. Enumerations in Swift are much more flexible, and don't have to provide a value for each case of the enumeration. If a value (known as a *raw* value) is provided for each enumeration case, the value can be a string, a character, or a value of any integer or floating-point type.

Alternatively, enumeration cases can specify associated values of *any* type to be stored along with each different case value, much as unions or variants do in other languages. You can define a common set of related cases as part of one enumeration, each of which has a different set of values of appropriate types associated with it.

Enumerations in Swift are first-class types in their own right. They adopt many features traditionally supported only by classes, such as computed properties to provide additional information about the enumeration's current value, and instance methods to provide functionality related to the values the enumeration represents. Enumerations can also define initializers to provide an initial case value; can be extended to expand their functionality beyond their original implementation; and can conform to protocols to provide standard functionality.

For more about these capabilities, see Properties, Methods, Initialization, Extensions, and Protocols.

Enumeration Syntax

You introduce enumerations with the `enum` keyword and place their entire definition within a pair of braces:

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

Here's an example for the four main points of a compass:

```
enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

The values defined in an enumeration (such as `north`, `south`, `east`, and `west`) are its *enumeration cases*. You use the `case` keyword to introduce new enumeration cases.

> NOTE
>
> Swift enumeration cases don't have an integer value set by default, unlike languages like C and Objective-C. In the `CompassPoint` example above, `north`, `south`, `east` and `west` don't implicitly equal `0`, `1`, `2` and `3`. Instead, the different enumeration cases are values in their own right, with an explicitly defined type of `CompassPoint`.

Multiple cases can appear on a single line, separated by commas:

```
enum Planet {
    case mercury, venus, earth, mars, jupiter,
     saturn, uranus, neptune
}
```

Each enumeration definition defines a new type. Like other types in Swift, their names (such as `CompassPoint` and `Planet`) start with a capital letter. Give enumeration types singular rather than plural names, so that they read as self-evident:

```
var directionToHead = CompassPoint.west
```

The type of `directionToHead` is inferred when it's initialized with one of the possible values of `CompassPoint`. Once `directionToHead` is declared as a `CompassPoint`, you can set it to a different `CompassPoint` value using a shorter dot syntax:

```
directionToHead = .east
```

The type of `directionToHead` is already known, and so you can drop the type when setting its value. This makes for highly readable code when working with explicitly typed enumeration values.

Matching Enumeration Values with a Switch Statement

You can match individual enumeration values with a `switch` statement:

```
directionToHead = .south
switch directionToHead {
case .north:
    print("Lots of planets have a north")
case .south:
    print("Watch out for penguins")
case .east:
    print("Where the sun rises")
case .west:
    print("Where the skies are blue")
}
// Prints "Watch out for penguins"
```

You can read this code as:

"Consider the value of `directionToHead`. In the case where it equals `.north`, print `"Lots of planets have a north"`. In the case where it equals `.south`, print `"Watch out for penguins"`."

…and so on.

As described in Control Flow, a `switch` statement must be exhaustive when considering an enumeration's cases. If the `case` for `.west` is omitted, this code doesn't compile, because it doesn't consider the complete list of `CompassPoint` cases. Requiring exhaustiveness ensures that enumeration cases aren't accidentally omitted.

When it isn't appropriate to provide a `case` for every enumeration case, you can provide a `default` case to cover any cases that aren't addressed explicitly:

```
let somePlanet = Planet.earth
switch somePlanet {
case .earth:
    print("Mostly harmless")
default:
    print("Not a safe place for humans")
}
// Prints "Mostly harmless"
```

Iterating over Enumeration Cases

For some enumerations, it's useful to have a collection of all of that enumeration's cases. You enable this by writing `: CaseIterable` after the enumeration's name. Swift exposes a collection of all the cases as an `allCases` property of the enumeration type. Here's an example:

```
enum Beverage: CaseIterable {
    case coffee, tea, juice
}
let numberOfChoices = Beverage.allCases.count
print("\(numberOfChoices) beverages available")
// Prints "3 beverages available"
```

In the example above, you write `Beverage.allCases` to access a collection that contains all of the cases of the `Beverage` enumeration. You can use `allCases` like any other collection—the collection's elements are instances of the enumeration type, so in this case they're `Beverage` values. The

example above counts how many cases there are, and the example below uses a `for-in` loop to iterate over all the cases.

```
for beverage in Beverage.allCases {
    print(beverage)
}
// coffee
// tea
// juice
```

The syntax used in the examples above marks the enumeration as conforming to the `CaseIterable` protocol. For information about protocols, see Protocols.

Associated Values

The examples in the previous section show how the cases of an enumeration are a defined (and typed) value in their own right. You can set a constant or variable to `Planet.earth`, and check for this value later. However, it's sometimes useful to be able to store values of other types alongside these case values. This additional information is called an *associated value*, and it varies each time you use that case as a value in your code.

You can define Swift enumerations to store associated values of any given type, and the value types can be different for each case of the enumeration if needed. Enumerations similar to these are known as *discriminated unions*, *tagged unions*, or *variants* in other programming languages.

For example, suppose an inventory tracking system needs to track products by two different types of barcode. Some products are labeled with 1D

barcodes in UPC format, which uses the numbers `0` to `9`. Each barcode has a number system digit, followed by five manufacturer code digits and five product code digits. These are followed by a check digit to verify that the code has been scanned correctly:

Other products are labeled with 2D barcodes in QR code format, which can use any ISO 8859-1 character and can encode a string up to 2,953 characters long:

It's convenient for an inventory tracking system to store UPC barcodes as a tuple of four integers, and QR code barcodes as a string of any length.

In Swift, an enumeration to define product barcodes of either type might look like this:

```
enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

This can be read as:

"Define an enumeration type called `Barcode`, which can take either a value of `upc` with an associated value of type (`Int`, `Int`, `Int`, `Int`), or a value of `qrCode` with an associated value of type `String`."

This definition doesn't provide any actual `Int` or `String` values—it just defines the *type* of associated values that `Barcode` constants and variables can store when they're equal to `Barcode.upc` or `Barcode.qrCode`.

You can then create new barcodes using either type:

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

This example creates a new variable called `productBarcode` and assigns it a value of `Barcode.upc` with an associated tuple value of `(8, 85909, 51226, 3)`.

You can assign the same product a different type of barcode:

```
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
```

At this point, the original `Barcode.upc` and its integer values are replaced by the new `Barcode.qrCode` and its string value. Constants and variables of type `Barcode` can store either a `.upc` or a `.qrCode` (together with their associated values), but they can store only one of them at any given time.

You can check the different barcode types using a switch statement, similar to the example in [Matching Enumeration Values with a Switch Statement]. This time, however, the associated values are extracted as part of the switch statement. You extract each associated value as a constant (with the `let` prefix) or a variable (with the `var` prefix) for use within the `switch` case's body:

```
switch productBarcode {
case .upc(let numberSystem, let manufacturer, let product, let check):
    print("UPC: \(numberSystem), \(manufacturer), \(product), \(check).")
case .qrCode(let productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

If all of the associated values for an enumeration case are extracted as constants, or if all are extracted as variables, you can place a single `var` or `let` annotation before the case name, for brevity:

```
switch productBarcode {
case let .upc(numberSystem, manufacturer, product, check):
    print("UPC : \(numberSystem), \(manufacturer), \(product), \(check).")
case let .qrCode(productCode):
    print("QR code: \(productCode).")
}
// Prints "QR code: ABCDEFGHIJKLMNOP."
```

Raw Values

The barcode example in Associated Values shows how cases of an enumeration can declare that they store associated values of different types. As an alternative to associated values, enumeration cases can come prepopulated with default values (called *raw values*), which are all of the same type.

Here's an example that stores raw ASCII values alongside named enumeration cases:

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

Here, the raw values for an enumeration called `ASCIIControlCharacter` are defined to be of type `Character`, and are set to some of the more common ASCII control characters. `Character` values are described in Strings and Characters.

Raw values can be strings, characters, or any of the integer or floating-point number types. Each raw value must be unique within its enumeration declaration.

> NOTE
>
> Raw values are *not* the same as associated values. Raw values are set to prepopulated values when you first define the enumeration in your code, like the three ASCII codes above. The raw value for a particular enumeration case is always the same. Associated values are set when you create a new constant or variable based on one of the enumeration's cases, and can be different each time you do so.

Implicitly Assigned Raw Values

When you're working with enumerations that store integer or string raw values, you don't have to explicitly assign a raw value for each case. When you don't, Swift automatically assigns the values for you.

For example, when integers are used for raw values, the implicit value for each case is one more than the previous case. If the first case doesn't have a value set, its value is `0`.

The enumeration below is a refinement of the earlier `Planet` enumeration, with integer raw values to represent each planet's order from the sun:

```
enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter,
     saturn, uranus, neptune
}
```

In the example above, `Planet.mercury` has an explicit raw value of `1`, `Planet.venus` has an implicit raw value of `2`, and so on.

When strings are used for raw values, the implicit value for each case is the text of that case's name.

The enumeration below is a refinement of the earlier `CompassPoint` enumeration, with string raw values to represent each direction's name:

```
enum CompassPoint: String {
    case north, south, east, west
}
```

In the example above, `CompassPoint.south` has an implicit raw value of `"south"`, and so on.

You access the raw value of an enumeration case with its `rawValue` property:

```
let earthsOrder = Planet.earth.rawValue
// earthsOrder is 3

let sunsetDirection = CompassPoint.west.rawValue
// sunsetDirection is "west"
```

Initializing from a Raw Value

If you define an enumeration with a raw-value type, the enumeration automatically receives an initializer that takes a value of the raw value's type (as a parameter called `rawValue`) and returns either an enumeration case or `nil`. You can use this initializer to try to create a new instance of the enumeration.

This example identifies Uranus from its raw value of `7`:

```
let possiblePlanet = Planet(rawValue: 7)
// possiblePlanet is of type Planet? and equals
     Planet.uranus
```

Not all possible `Int` values will find a matching planet, however. Because of this, the raw value initializer always returns an *optional* enumeration case. In the example above, `possiblePlanet` is of type `Planet?`, or "optional `Planet`."

> NOTE
>
> The raw value initializer is a failable initializer, because not every raw value will return an enumeration case. For more information, see Failable Initializers.

If you try to find a planet with a position of `11`, the optional `Planet` value returned by the raw value initializer will be `nil`:

```
let positionToFind = 11
if let somePlanet = Planet(rawValue: positionToFind) {
    switch somePlanet {
    case .earth:
        print("Mostly harmless")
    default:
        print("Not a safe place for humans")
    }
} else {
    print("There isn't a planet at position \(positionToFind)")
}
// Prints "There isn't a planet at position 11"
```

This example uses optional binding to try to access a planet with a raw value of `11`. The statement `if let somePlanet = Planet(rawValue: 11)` creates an optional `Planet`, and sets `somePlanet` to the value of that optional `Planet` if it can be retrieved. In this case, it isn't possible to retrieve a planet with a position of `11`, and so the `else` branch is executed instead.

Recursive Enumerations

A *recursive enumeration* is an enumeration that has another instance of the enumeration as the associated value for one or more of the enumeration cases. You indicate that an enumeration case is recursive by writing `indirect` before it, which tells the compiler to insert the necessary layer of indirection.

For example, here is an enumeration that stores simple arithmetic expressions:

```
enum ArithmeticExpression {
    case number(Int)
    indirect case addition(ArithmeticExpression,
     ArithmeticExpression)
    indirect case
     multiplication(ArithmeticExpression,
     ArithmeticExpression)
}
```

You can also write `indirect` before the beginning of the enumeration to enable indirection for all of the enumeration's cases that have an associated value:

```
indirect enum ArithmeticExpression {
    case number(Int)
    case addition(ArithmeticExpression,
     ArithmeticExpression)
    case multiplication(ArithmeticExpression,
     ArithmeticExpression)
}
```

This enumeration can store three kinds of arithmetic expressions: a plain number, the addition of two expressions, and the multiplication of two expressions. The `addition` and `multiplication` cases have associated values that are also arithmetic expressions—these associated values make it possible to nest expressions. For example, the expression `(5 + 4) * 2` has a number on the right-hand side of the multiplication and another expression on the left-hand side of the multiplication. Because the data is nested, the enumeration used to store the data also needs to support nesting—this means the enumeration needs to be recursive. The code below shows the `ArithmeticExpression` recursive enumeration being created for `(5 + 4) * 2`:

```
let five = ArithmeticExpression.number(5)
let four = ArithmeticExpression.number(4)
let sum = ArithmeticExpression.addition(five, four)
let product =
     ArithmeticExpression.multiplication(sum,
     ArithmeticExpression.number(2))
```

A recursive function is a straightforward way to work with data that has a recursive structure. For example, here's a function that evaluates an arithmetic expression:

```
func evaluate(_ expression: ArithmeticExpression) -> Int {
    switch expression {
    case let .number(value):
        return value
    case let .addition(left, right):
        return evaluate(left) + evaluate(right)
    case let .multiplication(left, right):
        return evaluate(left) * evaluate(right)
    }
}

print(evaluate(product))
// Prints "18"
```

This function evaluates a plain number by simply returning the associated value. It evaluates an addition or multiplication by evaluating the expression on the left-hand side, evaluating the expression on the right-hand side, and then adding them or multiplying them.

Structures and Classes

Structures and *classes* are general-purpose, flexible constructs that become the building blocks of your program's code. You define properties and methods to add functionality to your structures and classes using the same syntax you use to define constants, variables, and functions.

Unlike other programming languages, Swift doesn't require you to create separate interface and implementation files for custom structures and classes. In Swift, you define a structure or class in a single file, and the external interface to that class or structure is automatically made available for other code to use.

> NOTE
>
> An instance of a class is traditionally known as an *object*. However, Swift structures and classes are much closer in functionality than in other languages, and much of this chapter describes functionality that applies to instances of *either* a class or a structure type. Because of this, the more general term *instance* is used.

Comparing Structures and Classes

Structures and classes in Swift have many things in common. Both can:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state

- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

For more information, see Properties, Methods, Subscripts, Initialization, Extensions, and Protocols.

Classes have additional capabilities that structures don't have:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

For more information, see Inheritance, Type Casting, Deinitialization, and Automatic Reference Counting.

The additional capabilities that classes support come at the cost of increased complexity. As a general guideline, prefer structures because they're easier to reason about, and use classes when they're appropriate or necessary. In practice, this means most of the custom data types you define will be structures and enumerations. For a more detailed comparison, see Choosing Between Structures and Classes.

> NOTE
>
> Classes and actors share many of the same characteristics and behaviors. For information about actors, see Concurrency.

Definition Syntax

Structures and classes have a similar definition syntax. You introduce structures with the `struct` keyword and classes with the `class` keyword. Both place their entire definition within a pair of braces:

```
struct SomeStructure {
    // structure definition goes here
}
class SomeClass {
    // class definition goes here
}
```

> NOTE
>
> Whenever you define a new structure or class, you define a new Swift type. Give types `UpperCamelCase` names (such as `SomeStructure` and `SomeClass` here) to match the capitalization of standard Swift types (such as `String`, `Int`, and `Bool`). Give properties and methods `lowerCamelCase` names (such as `frameRate` and `incrementCount`) to differentiate them from type names.

Here's an example of a structure definition and a class definition:

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

The example above defines a new structure called `Resolution`, to describe a pixel-based display resolution. This structure has two stored properties called `width` and `height`. Stored properties are constants or variables that are bundled up and stored as part of the structure or class. These two properties are inferred to be of type `Int` by setting them to an initial integer value of `0`.

The example above also defines a new class called `VideoMode`, to describe a specific video mode for video display. This class has four variable stored properties. The first, `resolution`, is initialized with a new `Resolution` structure instance, which infers a property type of `Resolution`. For the other three properties, new `VideoMode` instances will be initialized with an `interlaced` setting of `false` (meaning "noninterlaced video"), a playback frame rate of `0.0`, and an optional `String` value called `name`. The `name` property is automatically given a default value of `nil`, or "no `name` value", because it's of an optional type.

Structure and Class Instances

The `Resolution` structure definition and the `VideoMode` class definition only describe what a `Resolution` or `VideoMode` will look like. They

themselves don't describe a specific resolution or video mode. To do that, you need to create an instance of the structure or class.

The syntax for creating instances is very similar for both structures and classes:

```
let someResolution = Resolution()
let someVideoMode = VideoMode()
```

Structures and classes both use initializer syntax for new instances. The simplest form of initializer syntax uses the type name of the class or structure followed by empty parentheses, such as `Resolution()` or `VideoMode()`. This creates a new instance of the class or structure, with any properties initialized to their default values. Class and structure initialization is described in more detail in [Initialization].

Accessing Properties

You can access the properties of an instance using *dot syntax*. In dot syntax, you write the property name immediately after the instance name, separated by a period (`.`), without any spaces:

```
print("The width of someResolution is \
      (someResolution.width)")
// Prints "The width of someResolution is 0"
```

In this example, `someResolution.width` refers to the `width` property of `someResolution`, and returns its default initial value of `0`.

You can drill down into subproperties, such as the `width` property in the `resolution` property of a `VideoMode`:

```
print("The width of someVideoMode is \
      (someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is 0"
```

You can also use dot syntax to assign a new value to a variable property:

```
someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \
      (someVideoMode.resolution.width)")
// Prints "The width of someVideoMode is now 1280"
```

Memberwise Initializers for Structure Types
All structures have an automatically generated *memberwise initializer*, which you can use to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name:

```
let vga = Resolution(width: 640, height: 480)
```

Unlike structures, class instances don't receive a default memberwise initializer. Initializers are described in more detail in Initialization.

Structures and Enumerations Are Value Types

A *value type* is a type whose value is *copied* when it's assigned to a variable or constant, or when it's passed to a function.

You've actually been using value types extensively throughout the previous chapters. In fact, all of the basic types in Swift—integers, floating-point numbers, Booleans, strings, arrays and dictionaries—are value types, and are implemented as structures behind the scenes.

All structures and enumerations are value types in Swift. This means that any structure and enumeration instances you create—and any value types they have as properties—are always copied when they're passed around in your code.

> NOTE
>
> Collections defined by the standard library like arrays, dictionaries, and strings use an optimization to reduce the performance cost of copying. Instead of making a copy immediately, these collections share the memory where the elements are stored between the original instance and any copies. If one of the copies of the collection is modified, the elements are copied just before the modification. The behavior you see in your code is always as if a copy took place immediately.

Consider this example, which uses the `Resolution` structure from the previous example:

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd
```

This example declares a constant called `hd` and sets it to a `Resolution` instance initialized with the width and height of full HD video (1920 pixels wide by 1080 pixels high).

It then declares a variable called `cinema` and sets it to the current value of `hd`. Because `Resolution` is a structure, a *copy* of the existing instance is made, and this new copy is assigned to `cinema`. Even though `hd` and `cinema` now have the same width and height, they're two completely different instances behind the scenes.

Next, the `width` property of `cinema` is amended to be the width of the slightly wider 2K standard used for digital cinema projection (2048 pixels

wide and 1080 pixels high):

```
cinema.width = 2048
```

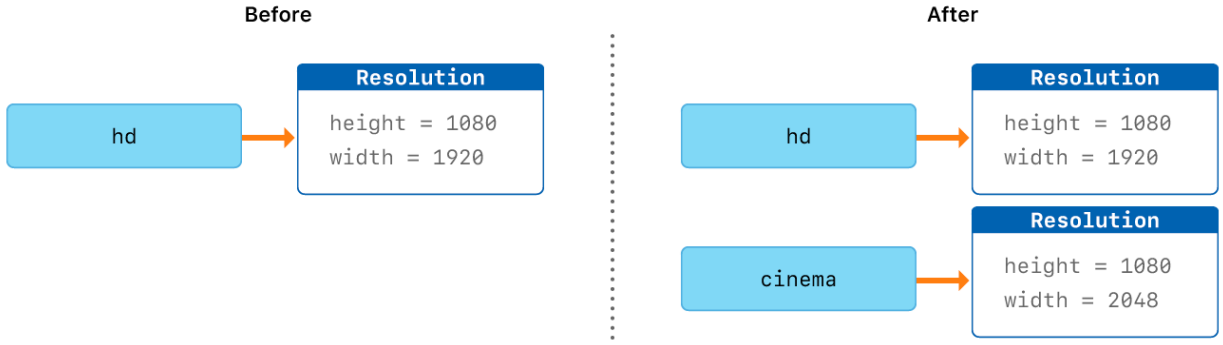
Checking the `width` property of `cinema` shows that it has indeed changed to be `2048`:

```
print("cinema is now \(cinema.width) pixels wide")
// Prints "cinema is now 2048 pixels wide"
```

However, the `width` property of the original `hd` instance still has the old value of `1920`:

```
print("hd is still \(hd.width) pixels wide")
// Prints "hd is still 1920 pixels wide"
```

When `cinema` was given the current value of `hd`, the *values* stored in `hd` were copied into the new `cinema` instance. The end result was two completely separate instances that contained the same numeric values. However, because they're separate instances, setting the width of `cinema` to `2048` doesn't affect the width stored in `hd`, as shown in the figure below:



The same behavior applies to enumerations:

```swift
enum CompassPoint {
    case north, south, east, west
    mutating func turnNorth() {
        self = .north
    }
}
var currentDirection = CompassPoint.west
let rememberedDirection = currentDirection
currentDirection.turnNorth()

print("The current direction is \(currentDirection)")
print("The remembered direction is \
     (rememberedDirection)")
// Prints "The current direction is north"
// Prints "The remembered direction is west"
```

When `rememberedDirection` is assigned the value of `currentDirection`, it's actually set to a copy of that value. Changing the value of `currentDirection` thereafter doesn't affect the copy of the original value that was stored in `rememberedDirection`.

Classes Are Reference Types

Unlike value types, *reference types* are *not* copied when they're assigned to a variable or constant, or when they're passed to a function. Rather than a copy, a reference to the same existing instance is used.

Here's an example, using the `VideoMode` class defined above:

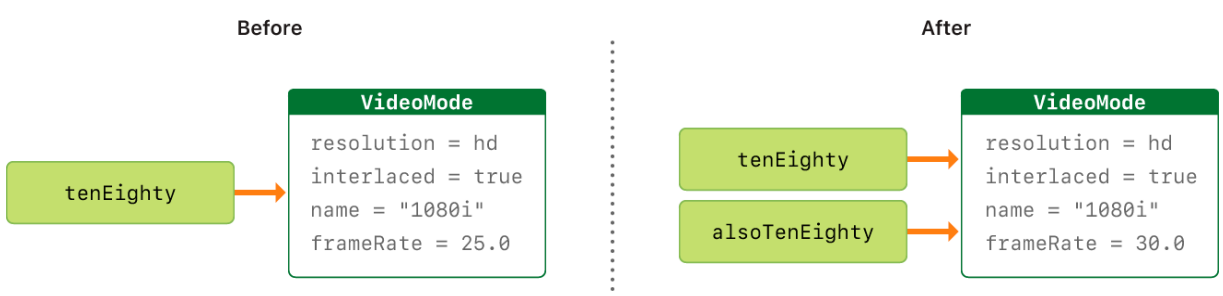
```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0
```

This example declares a new constant called `tenEighty` and sets it to refer to a new instance of the `VideoMode` class. The video mode is assigned a copy of the HD resolution of `1920` by `1080` from before. It's set to be interlaced, its name is set to `"1080i"`, and its frame rate is set to `25.0` frames per second.

Next, `tenEighty` is assigned to a new constant, called `alsoTenEighty`, and the frame rate of `alsoTenEighty` is modified:

```
let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0
```

Because classes are reference types, `tenEighty` and `alsoTenEighty` actually both refer to the *same* `VideoMode` instance. Effectively, they're just two different names for the same single instance, as shown in the figure below:



Checking the `frameRate` property of `tenEighty` shows that it correctly reports the new frame rate of `30.0` from the underlying `VideoMode` instance:

```
print("The frameRate property of tenEighty is now \
      (tenEighty.frameRate)")
// Prints "The frameRate property of tenEighty is now
      30.0"
```

This example also shows how reference types can be harder to reason about. If `tenEighty` and `alsoTenEighty` were far apart in your program's code, it could be difficult to find all the ways that the video mode is changed. Wherever you use `tenEighty`, you also have to think about the code that uses `alsoTenEighty`, and vice versa. In contrast, value types are easier to reason about because all of the code that interacts with the same value is close together in your source files.

Note that `tenEighty` and `alsoTenEighty` are declared as *constants*, rather than variables. However, you can still change `tenEighty.frameRate` and `alsoTenEighty.frameRate` because the values of the `tenEighty` and `alsoTenEighty` constants themselves don't actually change. `tenEighty` and `alsoTenEighty` themselves don't "store" the `VideoMode` instance—instead, they both *refer* to a `VideoMode` instance behind the scenes. It's the `frameRate` property of the underlying `VideoMode` that's changed, not the values of the constant references to that `VideoMode`.

Identity Operators

Because classes are reference types, it's possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same isn't true for structures and enumerations, because they're always copied when they're assigned to a constant or variable, or passed to a function.)

It can sometimes be useful to find out whether two constants or variables refer to exactly the same instance of a class. To enable this, Swift provides two identity operators:

- Identical to (`===`)
- Not identical to (`!==`)

Use these operators to check whether two constants or variables refer to the same single instance:

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
}
// Prints "tenEighty and alsoTenEighty refer to the same VideoMode instance."
```

Note that *identical to* (represented by three equals signs, or `===`) doesn't mean the same thing as *equal to* (represented by two equals signs, or `==`). *Identical to* means that two constants or variables of class type refer to exactly the same class instance. *Equal to* means that two instances are considered equal or equivalent in value, for some appropriate meaning of *equal*, as defined by the type's designer.

When you define your own custom structures and classes, it's your responsibility to decide what qualifies as two instances being equal. The process of defining your own implementations of the `==` and `!=` operators is described in Equivalence Operators.

Pointers

If you have experience with C, C++, or Objective-C, you may know that these languages use *pointers* to refer to addresses in memory. A Swift

constant or variable that refers to an instance of some reference type is similar to a pointer in C, but isn't a direct pointer to an address in memory, and doesn't require you to write an asterisk (*) to indicate that you are creating a reference. Instead, these references are defined like any other constant or variable in Swift. The standard library provides pointer and buffer types that you can use if you need to interact with pointers directly—see Manual Memory Management.

Converted by Evan at Apps Dissected -www.appsdissected.com

Properties

Properties associate values with a particular class, structure, or enumeration. Stored properties store constant and variable values as part of an instance, whereas computed properties calculate (rather than store) a value. Computed properties are provided by classes, structures, and enumerations. Stored properties are provided only by classes and structures.

Stored and computed properties are usually associated with instances of a particular type. However, properties can also be associated with the type itself. Such properties are known as type properties.

In addition, you can define property observers to monitor changes in a property's value, which you can respond to with custom actions. Property observers can be added to stored properties you define yourself, and also to properties that a subclass inherits from its superclass.

You can also use a property wrapper to reuse code in the getter and setter of multiple properties.

Stored Properties

In its simplest form, a stored property is a constant or variable that's stored as part of an instance of a particular class or structure. Stored properties can be either *variable stored properties* (introduced by the `var` keyword) or *constant stored properties* (introduced by the `let` keyword).

You can provide a default value for a stored property as part of its definition, as described in Default Property Values. You can also set and modify the initial value for a stored property during initialization. This is true even for constant stored properties, as described in Assigning Constant Properties During Initialization.

The example below defines a structure called `FixedLengthRange`, which describes a range of integers whose range length can't be changed after it's created:

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

Instances of `FixedLengthRange` have a variable stored property called `firstValue` and a constant stored property called `length`. In the example above, `length` is initialized when the new range is created and can't be changed thereafter, because it's a constant property.

Stored Properties of Constant Structure Instances

If you create an instance of a structure and assign that instance to a constant, you can't modify the instance's properties, even if they were declared as variable properties:

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even though firstValue is a variable property
```

Because `rangeOfFourItems` is declared as a constant (with the `let` keyword), it isn't possible to change its `firstValue` property, even though `firstValue` is a variable property.

This behavior is due to structures being *value types*. When an instance of a value type is marked as a constant, so are all of its properties.

The same isn't true for classes, which are *reference types*. If you assign an instance of a reference type to a constant, you can still change that instance's variable properties.

Lazy Stored Properties

A *lazy stored property* is a property whose initial value isn't calculated until the first time it's used. You indicate a lazy stored property by writing the `lazy` modifier before its declaration.

> NOTE
>
> You must always declare a lazy property as a variable (with the `var` keyword), because its initial value might not be retrieved until after instance initialization completes. Constant properties must always have a value *before* initialization completes, and therefore can't be declared as lazy.

Lazy properties are useful when the initial value for a property is dependent on outside factors whose values aren't known until after an instance's initialization is complete. Lazy properties are also useful when the initial value for a property requires complex or computationally expensive setup that shouldn't be performed unless or until it's needed.

The example below uses a lazy stored property to avoid unnecessary initialization of a complex class. This example defines two classes called `DataImporter` and `DataManager`, neither of which is shown in full:

```
class DataImporter {
    /*
    DataImporter is a class to import data from an external file.
    The class is assumed to take a nontrivial amount of time to initialize.
    */
    var filename = "data.txt"
    // the DataImporter class would provide data importing functionality here
}

class DataManager {
    lazy var importer = DataImporter()
    var data: [String] = []
    // the DataManager class would provide data management functionality here
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
// the DataImporter instance for the importer property hasn't yet been created
```

The `DataManager` class has a stored property called `data`, which is initialized with a new, empty array of `String` values. Although the rest of its functionality isn't shown, the purpose of this `DataManager` class is to manage and provide access to this array of `String` data.

Part of the functionality of the `DataManager` class is the ability to import data from a file. This functionality is provided by the `DataImporter` class, which is assumed to take a nontrivial amount of time to initialize. This might be because a `DataImporter` instance needs to open a file and read its contents into memory when the `DataImporter` instance is initialized.

Because it's possible for a `DataManager` instance to manage its data without ever importing data from a file, `DataManager` doesn't create a new `DataImporter` instance when the `DataManager` itself is created. Instead, it makes more sense to create the `DataImporter` instance if and when it's first used.

Because it's marked with the `lazy` modifier, the `DataImporter` instance for the `importer` property is only created when the `importer` property is first accessed, such as when its `filename` property is queried:

```
print(manager.importer.filename)
// the DataImporter instance for the importer
     property has now been created
// Prints "data.txt"
```

> NOTE
>
> If a property marked with the `lazy` modifier is accessed by multiple threads simultaneously and the property hasn't yet been initialized, there's no guarantee that the property will be initialized only once.

Stored Properties and Instance Variables

If you have experience with Objective-C, you may know that it provides *two* ways to store values and references as part of a class instance. In addition to properties, you can use instance variables as a backing store for the values stored in a property.

Swift unifies these concepts into a single property declaration. A Swift property doesn't have a corresponding instance variable, and the backing store for a property isn't accessed directly. This approach avoids confusion about how the value is accessed in different contexts and simplifies the property's declaration into a single, definitive statement. All information about the property—including its name, type, and memory management characteristics—is defined in a single location as part of the type's definition.

Computed Properties

In addition to stored properties, classes, structures, and enumerations can define *computed properties*, which don't actually store a value. Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly.

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
```

```
                    size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// Prints "square.origin is now at (10.0, 10.0)"
```

This example defines three structures for working with geometric shapes:

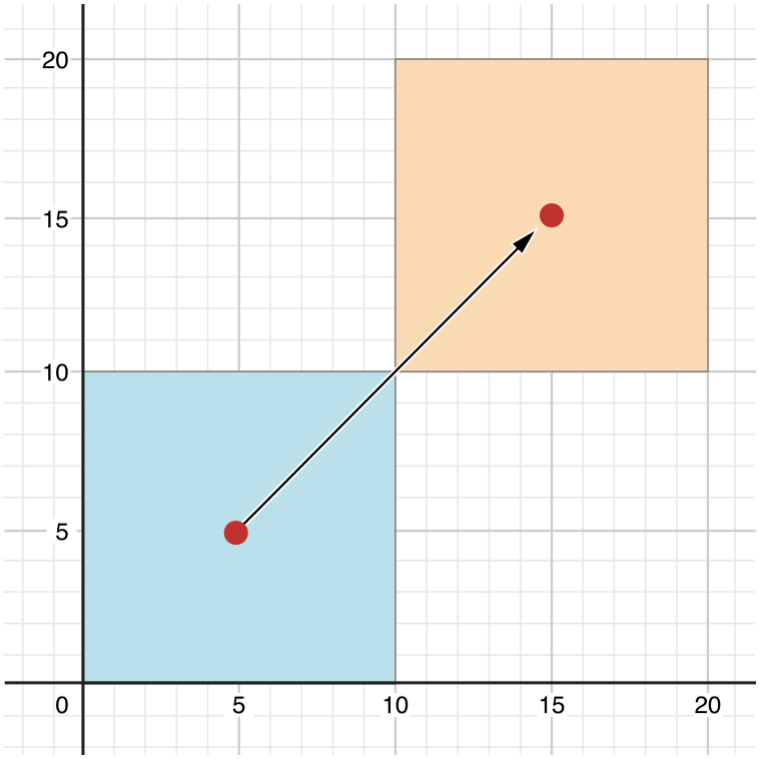
- `Point` encapsulates the x- and y-coordinate of a point.
- `Size` encapsulates a `width` and a `height`.
- `Rect` defines a rectangle by an origin point and a size.

The `Rect` structure also provides a computed property called `center`. The current center position of a `Rect` can always be determined from its `origin` and `size`, and so you don't need to store the center point as an explicit `Point` value. Instead, `Rect` defines a custom getter and setter for a computed variable called `center`, to enable you to work with the rectangle's `center` as if it were a real stored property.

The example above creates a new `Rect` variable called `square`. The `square` variable is initialized with an origin point of `(0, 0)`, and a width and height of `10`. This square is represented by the blue square in the diagram below.

The `square` variable's `center` property is then accessed through dot syntax (`square.center`), which causes the getter for `center` to be called, to retrieve the current property value. Rather than returning an existing value, the getter actually calculates and returns a new `Point` to represent the center of the square. As can be seen above, the getter correctly returns a center point of `(5, 5)`.

The `center` property is then set to a new value of `(15, 15)`, which moves the square up and to the right, to the new position shown by the orange square in the diagram below. Setting the `center` property calls the setter for `center`, which modifies the `x` and `y` values of the stored `origin` property, and moves the square to its new position.



Shorthand Setter Declaration

If a computed property's setter doesn't define a name for the new value to be set, a default name of `newValue` is used. Here's an alternative version of the `Rect` structure that takes advantage of this shorthand notation:

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

Shorthand Getter Declaration

If the entire body of a getter is a single expression, the getter implicitly returns that expression. Here's an another version of the `Rect` structure that takes advantage of this shorthand notation and the shorthand notation for setters:

```
struct CompactRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            Point(x: origin.x + (size.width / 2),
                  y: origin.y + (size.height / 2))
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

Omitting the `return` from a getter follows the same rules as omitting `return` from a function, as described in [Functions With an Implicit Return](#).

Read-Only Computed Properties

A computed property with a getter but no setter is known as a *read-only computed property*. A read-only computed property always returns a value, and can be accessed through dot syntax, but can't be set to a different value.

> NOTE
>
> You must declare computed properties—including read-only computed properties—as variable properties with the `var` keyword, because their value isn't fixed. The `let` keyword is only used for constant properties, to indicate that their values can't be changed once they're set as part of instance initialization.

You can simplify the declaration of a read-only computed property by removing the `get` keyword and its braces:

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height:
     5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \
     (fourByFiveByTwo.volume)")
// Prints "the volume of fourByFiveByTwo is 40.0"
```

This example defines a new structure called `Cuboid`, which represents a 3D rectangular box with `width`, `height`, and `depth` properties. This structure also has a read-only computed property called `volume`, which calculates and returns the current volume of the cuboid. It doesn't make sense for `volume` to be settable, because it would be ambiguous as to which values of `width`, `height`, and `depth` should be used for a particular `volume` value. Nonetheless, it's useful for a `Cuboid` to provide a read-only computed property to enable external users to discover its current calculated volume.

Property Observers

Property observers observe and respond to changes in a property's value. Property observers are called every time a property's value is set, even if the new value is the same as the property's current value.

You can add property observers in the following places:

- Stored properties that you define
- Stored properties that you inherit
- Computed properties that you inherit

For an inherited property, you add a property observer by overriding that property in a subclass. For a computed property that you define, use the property's setter to observe and respond to value changes, instead of trying to create an observer. Overriding properties is described in Overriding.

You have the option to define either or both of these observers on a property:

- `willSet` is called just before the value is stored.
- `didSet` is called immediately after the new value is stored.

If you implement a `willSet` observer, it's passed the new property value as a constant parameter. You can specify a name for this parameter as part of your `willSet` implementation. If you don't write the parameter name and parentheses within your implementation, the parameter is made available with a default parameter name of `newValue`.

Similarly, if you implement a `didSet` observer, it's passed a constant parameter containing the old property value. You can name the parameter or use the default parameter name of `oldValue`. If you assign a value to a property within its own `didSet` observer, the new value that you assign replaces the one that was just set.

> NOTE
>
> The `willSet` and `didSet` observers of superclass properties are called when a property is set in a subclass initializer, after the superclass initializer has been called. They aren't called while a class is setting its own properties, before the superclass initializer has been called.
>
> For more information about initializer delegation, see Initializer Delegation for Value Types and Initializer Delegation for Class Types.

Here's an example of `willSet` and `didSet` in action. The example below defines a new class called `StepCounter`, which tracks the total number of steps that a person takes while walking. This class might be used with input data from a pedometer or other step counter to keep track of a person's exercise during their daily routine.

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

The `StepCounter` class declares a `totalSteps` property of type `Int`. This is a stored property with `willSet` and `didSet` observers.

The `willSet` and `didSet` observers for `totalSteps` are called whenever the property is assigned a new value. This is true even if the new value is the same as the current value.

This example's `willSet` observer uses a custom parameter name of `newTotalSteps` for the upcoming new value. In this example, it simply prints out the value that's about to be set.

The `didSet` observer is called after the value of `totalSteps` is updated. It compares the new value of `totalSteps` against the old value. If the total number of steps has increased, a message is printed to indicate how many new steps have been taken. The `didSet` observer doesn't provide a custom parameter name for the old value, and the default name of `oldValue` is used instead.

> NOTE
>
> If you pass a property that has observers to a function as an in-out parameter, the `willSet` and `didSet` observers are always called. This is because of the copy-in copy-out memory model for in-out parameters: The value is always written back to the property at the end of the function. For a detailed discussion of the behavior of in-out parameters, see In-Out Parameters.

Property Wrappers

A property wrapper adds a layer of separation between code that manages how a property is stored and the code that defines a property. For example, if you have properties that provide thread-safety checks or store their underlying data in a database, you have to write that code on every property. When you use a property wrapper, you write the management

code once when you define the wrapper, and then reuse that management code by applying it to multiple properties.

To define a property wrapper, you make a structure, enumeration, or class that defines a `wrappedValue` property. In the code below, the `TwelveOrLess` structure ensures that the value it wraps always contains a number less than or equal to 12. If you ask it to store a larger number, it stores 12 instead.

```
@propertyWrapper
struct TwelveOrLess {
    private var number = 0
    var wrappedValue: Int {
        get { return number }
        set { number = min(newValue, 12) }
    }
}
```

The setter ensures that new values are less than 12, and the getter returns the stored value.

> NOTE
>
> The declaration for `number` in the example above marks the variable as `private`, which ensures `number` is used only in the implementation of `TwelveOrLess`. Code that's written anywhere else accesses the value using the getter and setter for `wrappedValue`, and can't use `number` directly. For information about `private`, see Access Control.

You apply a wrapper to a property by writing the wrapper's name before the property as an attribute. Here's a structure that stores a rectangle that uses the `TwelveOrLess` property wrapper to ensure its dimensions are always 12 or less:

```
struct SmallRectangle {
    @TwelveOrLess var height: Int
    @TwelveOrLess var width: Int
}

var rectangle = SmallRectangle()
print(rectangle.height)
// Prints "0"

rectangle.height = 10
print(rectangle.height)
// Prints "10"

rectangle.height = 24
print(rectangle.height)
// Prints "12"
```

The `height` and `width` properties get their initial values from the definition of `TwelveOrLess`, which sets `TwelveOrLess.number` to zero. The setter in `TwelveOrLess` treats 10 as a valid value so storing the number 10 in `rectangle.height` proceeds as written. However, 24 is larger than `TwelveOrLess` allows, so trying to store 24 end up setting `rectangle.height` to 12 instead, the largest allowed value.

When you apply a wrapper to a property, the compiler synthesizes code that provides storage for the wrapper and code that provides access to the property through the wrapper. (The property wrapper is responsible for storing the wrapped value, so there's no synthesized code for that.) You could write code that uses the behavior of a property wrapper, without taking advantage of the special attribute syntax. For example, here's a

version of `SmallRectangle` from the previous code listing that wraps its properties in the `TwelveOrLess` structure explicitly, instead of writing `@TwelveOrLess` as an attribute:

```
struct SmallRectangle {
    private var _height = TwelveOrLess()
    private var _width = TwelveOrLess()
    var height: Int {
        get { return _height.wrappedValue }
        set { _height.wrappedValue = newValue }
    }
    var width: Int {
        get { return _width.wrappedValue }
        set { _width.wrappedValue = newValue }
    }
}
```

The `_height` and `_width` properties store an instance of the property wrapper, `TwelveOrLess`. The getter and setter for `height` and `width` wrap access to the `wrappedValue` property.

Setting Initial Values for Wrapped Properties

The code in the examples above sets the initial value for the wrapped property by giving `number` an initial value in the definition of `TwelveOrLess`. Code that uses this property wrapper, can't specify a different initial value for a property that's wrapped by `TwelveOrLess`—for example, the definition of `SmallRectangle` can't give `height` or `width` initial values. To support setting an initial value or other customization, the property wrapper needs to add an initializer. Here's an expanded version of

`TwelveOrLess` called `SmallNumber` that defines initializers that set the wrapped and maximum value:

```
@propertyWrapper
struct SmallNumber {
    private var maximum: Int
    private var number: Int

    var wrappedValue: Int {
        get { return number }
        set { number = min(newValue, maximum) }
    }

    init() {
        maximum = 12
        number = 0
    }
    init(wrappedValue: Int) {
        maximum = 12
        number = min(wrappedValue, maximum)
    }
    init(wrappedValue: Int, maximum: Int) {
        self.maximum = maximum
        number = min(wrappedValue, maximum)
    }
}
```

The definition of `SmallNumber` includes three initializers — `init()`, `init(wrappedValue:)`, and `init(wrappedValue:maximum:)` — which the examples below use to set the wrapped value and the maximum value. For information about initialization and initializer syntax, see Initialization.

When you apply a wrapper to a property and you don't specify an initial value, Swift uses the `init()` initializer to set up the wrapper. For example:

```
struct ZeroRectangle {
    @SmallNumber var height: Int
    @SmallNumber var width: Int
}

var zeroRectangle = ZeroRectangle()
print(zeroRectangle.height, zeroRectangle.width)
// Prints "0 0"
```

The instances of `SmallNumber` that wrap `height` and `width` are created by calling `SmallNumber()`. The code inside that initializer sets the initial wrapped value and the initial maximum value, using the default values of zero and 12. The property wrapper still provides all of the initial values, like the earlier example that used `TwelveOrLess` in `SmallRectangle`. Unlike that example, `SmallNumber` also supports writing those initial values as part of declaring the property.

When you specify an initial value for the property, Swift uses the `init(wrappedValue:)` initializer to set up the wrapper. For example:

```swift
struct UnitRectangle {
    @SmallNumber var height: Int = 1
    @SmallNumber var width: Int = 1
}

var unitRectangle = UnitRectangle()
print(unitRectangle.height, unitRectangle.width)
// Prints "1 1"
```

When you write `= 1` on a property with a wrapper, that's translated into a call to the `init(wrappedValue:)` initializer. The instances of `SmallNumber` that wrap `height` and `width` are created by calling `SmallNumber(wrappedValue: 1)`. The initializer uses the wrapped value that's specified here, and it uses the default maximum value of 12.

When you write arguments in parentheses after the custom attribute, Swift uses the initializer that accepts those arguments to set up the wrapper. For example, if you provide an initial value and a maximum value, Swift uses the `init(wrappedValue:maximum:)` initializer:

```
struct NarrowRectangle {
    @SmallNumber(wrappedValue: 2, maximum: 5) var height: Int
    @SmallNumber(wrappedValue: 3, maximum: 4) var width: Int
}

var narrowRectangle = NarrowRectangle()
print(narrowRectangle.height, narrowRectangle.width)
// Prints "2 3"

narrowRectangle.height = 100
narrowRectangle.width = 100
print(narrowRectangle.height, narrowRectangle.width)
// Prints "5 4"
```

The instance of `SmallNumber` that wraps `height` is created by calling `SmallNumber(wrappedValue: 2, maximum: 5)`, and the instance that wraps `width` is created by calling `SmallNumber(wrappedValue: 3, maximum: 4)`.

By including arguments to the property wrapper, you can set up the initial state in the wrapper or pass other options to the wrapper when it's created. This syntax is the most general way to use a property wrapper. You can provide whatever arguments you need to the attribute, and they're passed to the initializer.

When you include property wrapper arguments, you can also specify an initial value using assignment. Swift treats the assignment like a

`wrappedValue` argument and uses the initializer that accepts the arguments you include. For example:

```
struct MixedRectangle {
    @SmallNumber var height: Int = 1
    @SmallNumber(maximum: 9) var width: Int = 2
}

var mixedRectangle = MixedRectangle()
print(mixedRectangle.height)
// Prints "1"

mixedRectangle.height = 20
print(mixedRectangle.height)
// Prints "12"
```

The instance of `SmallNumber` that wraps `height` is created by calling `SmallNumber(wrappedValue: 1)`, which uses the default maximum value of 12. The instance that wraps `width` is created by calling `SmallNumber(wrappedValue: 2, maximum: 9)`.

Projecting a Value From a Property Wrapper

In addition to the wrapped value, a property wrapper can expose additional functionality by defining a *projected value*—for example, a property wrapper that manages access to a database can expose a `flushDatabaseConnection()` method on its projected value. The name of the projected value is the same as the wrapped value, except it begins with a dollar sign (`$`). Because your code can't define properties that start with `$` the projected value never interferes with properties you define.

In the `SmallNumber` example above, if you try to set the property to a number that's too large, the property wrapper adjusts the number before storing it. The code below adds a `projectedValue` property to the `SmallNumber` structure to keep track of whether the property wrapper adjusted the new value for the property before storing that new value.

```
@propertyWrapper
struct SmallNumber {
    private var number: Int
    private(set) var projectedValue: Bool

    var wrappedValue: Int {
        get { return number }
        set {
            if newValue > 12 {
                number = 12
                projectedValue = true
            } else {
                number = newValue
                projectedValue = false
            }
        }
    }

    init() {
        self.number = 0
        self.projectedValue = false
    }
}
struct SomeStructure {
    @SmallNumber var someNumber: Int
}
```

```
27  var someStructure = SomeStructure()
28
29  someStructure.someNumber = 4
30  print(someStructure.$someNumber)
31  // Prints "false"
32
33  someStructure.someNumber = 55
34  print(someStructure.$someNumber)
35  // Prints "true"
```

Writing `someStructure.$someNumber` accesses the wrapper's projected value. After storing a small number like four, the value of `someStructure.$someNumber` is `false`. However, the projected value is `true` after trying to store a number that's too large, like 55.

A property wrapper can return a value of any type as its projected value. In this example, the property wrapper exposes only one piece of information—whether the number was adjusted—so it exposes that Boolean value as its projected value. A wrapper that needs to expose more information can return an instance of some other data type, or it can return `self` to expose the instance of the wrapper as its projected value.

When you access a projected value from code that's part of the type, like a property getter or an instance method, you can omit `self.` before the property name, just like accessing other properties. The code in the following example refers to the projected value of the wrapper around `height` and `width` as `$height` and `$width`:

```
enum Size {
    case small, large
}

struct SizedRectangle {
    @SmallNumber var height: Int
    @SmallNumber var width: Int

    mutating func resize(to size: Size) -> Bool {
        switch size {
        case .small:
            height = 10
            width = 20
        case .large:
            height = 100
            width = 100
        }
        return $height || $width
    }
}
```

Because property wrapper syntax is just syntactic sugar for a property with a getter and a setter, accessing `height` and `width` behaves the same as accessing any other property. For example, the code in `resize(to:)` accesses `height` and `width` using their property wrapper. If you call `resize(to: .large)`, the switch case for `.large` sets the rectangle's height and width to 100. The wrapper prevents the value of those properties from being larger than 12, and it sets the projected value to `true`, to record

the fact that it adjusted their values. At the end of `resize(to:)`, the return statement checks `$height` and `$width` to determine whether the property wrapper adjusted either `height` or `width`.

Global and Local Variables

The capabilities described above for computing and observing properties are also available to *global variables* and *local variables*. Global variables are variables that are defined outside of any function, method, closure, or type context. Local variables are variables that are defined within a function, method, or closure context.

The global and local variables you have encountered in previous chapters have all been *stored variables*. Stored variables, like stored properties, provide storage for a value of a certain type and allow that value to be set and retrieved.

However, you can also define *computed variables* and define observers for stored variables, in either a global or local scope. Computed variables calculate their value, rather than storing it, and they're written in the same way as computed properties.

> NOTE
>
> Global constants and variables are always computed lazily, in a similar manner to Lazy Stored Properties. Unlike lazy stored properties, global constants and variables don't need to be marked with the `lazy` modifier.
>
> Local constants and variables are never computed lazily.

You can apply a property wrapper to a local stored variable, but not to a global variable or a computed variable. For example, in the code below, `myNumber` uses `SmallNumber` as a property wrapper.

```
1  func someFunction() {
2      @SmallNumber var myNumber: Int = 0
3
4      myNumber = 10
5      // now myNumber is 10
6
7      myNumber = 24
8      // now myNumber is 12
9  }
```

Like when you apply `SmallNumber` to a property, setting the value of `myNumber` to 10 is valid. Because the property wrapper doesn't allow values higher than 12, it sets `myNumber` to 12 instead of 24.

Type Properties

Instance properties are properties that belong to an instance of a particular type. Every time you create a new instance of that type, it has its own set of property values, separate from any other instance.

You can also define properties that belong to the type itself, not to any one instance of that type. There will only ever be one copy of these properties, no matter how many instances of that type you create. These kinds of properties are called *type properties*.

Type properties are useful for defining values that are universal to *all* instances of a particular type, such as a constant property that all instances can use (like a static constant in C), or a variable property that stores a value that's global to all instances of that type (like a static variable in C).

Stored type properties can be variables or constants. Computed type properties are always declared as variable properties, in the same way as computed instance properties.

> NOTE
>
> Unlike stored instance properties, you must always give stored type properties a default value. This is because the type itself doesn't have an initializer that can assign a value to a stored type property at initialization time.
>
> Stored type properties are lazily initialized on their first access. They're guaranteed to be initialized only once, even when accessed by multiple threads simultaneously, and they don't need to be marked with the `lazy` modifier.

Type Property Syntax

In C and Objective-C, you define static constants and variables associated with a type as *global* static variables. In Swift, however, type properties are written as part of the type's definition, within the type's outer curly braces, and each type property is explicitly scoped to the type it supports.

You define type properties with the `static` keyword. For computed type properties for class types, you can use the `class` keyword instead to allow subclasses to override the superclass's implementation. The example below shows the syntax for stored and computed type properties:

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int
     {
        return 107
    }
}
```

> NOTE
>
> The computed type property examples above are for read-only computed type properties, but you can also define read-write computed type properties with the same syntax as for computed instance properties.

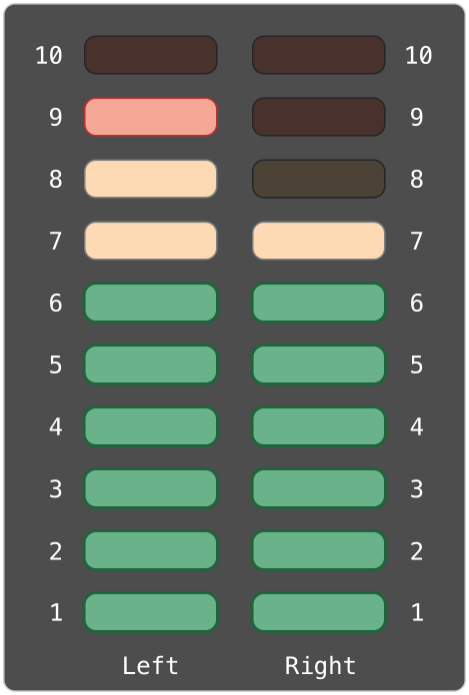
Querying and Setting Type Properties

Type properties are queried and set with dot syntax, just like instance properties. However, type properties are queried and set on the *type*, not on an instance of that type. For example:

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value."
print(SomeEnumeration.computedTypeProperty)
// Prints "6"
print(SomeClass.computedTypeProperty)
// Prints "27"
```

The examples that follow use two stored type properties as part of a structure that models an audio level meter for a number of audio channels. Each channel has an integer audio level between `0` and `10` inclusive.

The figure below illustrates how two of these audio channels can be combined to model a stereo audio level meter. When a channel's audio level is `0`, none of the lights for that channel are lit. When the audio level is `10`, all of the lights for that channel are lit. In this figure, the left channel has a current level of `9`, and the right channel has a current level of `7`:



The audio channels described above are represented by instances of the `AudioChannel` structure:

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel > AudioChannel.thresholdLevel {
                // cap the new audio level to the threshold level
                currentLevel = AudioChannel.thresholdLevel
            }
            if currentLevel > AudioChannel.maxInputLevelForAllChannels {
                // store this as the new overall maximum input level

                AudioChannel.maxInputLevelForAllChannels = currentLevel
            }
        }
    }
}
```

The `AudioChannel` structure defines two stored type properties to support its functionality. The first, `thresholdLevel`, defines the maximum threshold value an audio level can take. This is a constant value of `10` for all

`AudioChannel` instances. If an audio signal comes in with a higher value than `10`, it will be capped to this threshold value (as described below).

The second type property is a variable stored property called `maxInputLevelForAllChannels`. This keeps track of the maximum input value that has been received by *any* `AudioChannel` instance. It starts with an initial value of `0`.

The `AudioChannel` structure also defines a stored instance property called `currentLevel`, which represents the channel's current audio level on a scale of `0` to `10`.

The `currentLevel` property has a `didSet` property observer to check the value of `currentLevel` whenever it's set. This observer performs two checks:

- If the new value of `currentLevel` is greater than the allowed `thresholdLevel`, the property observer caps `currentLevel` to `thresholdLevel`.
- If the new value of `currentLevel` (after any capping) is higher than any value previously received by *any* `AudioChannel` instance, the property observer stores the new `currentLevel` value in the `maxInputLevelForAllChannels` type property.

> NOTE
>
> In the first of these two checks, the `didSet` observer sets `currentLevel` to a different value. This doesn't, however, cause the observer to be called again.

You can use the `AudioChannel` structure to create two new audio channels called `leftChannel` and `rightChannel`, to represent the audio levels of a stereo sound system:

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

If you set the `currentLevel` of the *left* channel to `7`, you can see that the `maxInputLevelForAllChannels` type property is updated to equal `7`:

```
leftChannel.currentLevel = 7
print(leftChannel.currentLevel)
// Prints "7"
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "7"
```

If you try to set the `currentLevel` of the *right* channel to `11`, you can see that the right channel's `currentLevel` property is capped to the maximum value of `10`, and the `maxInputLevelForAllChannels` type property is updated to equal `10`:

```
rightChannel.currentLevel = 11
print(rightChannel.currentLevel)
// Prints "10"
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "10"
```

Methods

Methods are functions that are associated with a particular type. Classes, structures, and enumerations can all define instance methods, which encapsulate specific tasks and functionality for working with an instance of a given type. Classes, structures, and enumerations can also define type methods, which are associated with the type itself. Type methods are similar to class methods in Objective-C.

The fact that structures and enumerations can define methods in Swift is a major difference from C and Objective-C. In Objective-C, classes are the only types that can define methods. In Swift, you can choose whether to define a class, structure, or enumeration, and still have the flexibility to define methods on the type you create.

Instance Methods

Instance methods are functions that belong to instances of a particular class, structure, or enumeration. They support the functionality of those instances, either by providing ways to access and modify instance properties, or by providing functionality related to the instance's purpose. Instance methods have exactly the same syntax as functions, as described in Functions.

You write an instance method within the opening and closing braces of the type it belongs to. An instance method has implicit access to all other instance methods and properties of that type. An instance method can be called only on a specific instance of the type it belongs to. It can't be called in isolation without an existing instance.

Here's an example that defines a simple `Counter` class, which can be used to count the number of times an action occurs:

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func increment(by amount: Int) {
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

The `Counter` class defines three instance methods:

- `increment()` increments the counter by `1`.
- `increment(by: Int)` increments the counter by a specified integer amount.
- `reset()` resets the counter to zero.

The `Counter` class also declares a variable property, `count`, to keep track of the current counter value.

You call instance methods with the same dot syntax as properties:

```
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.increment(by: 5)
// the counter's value is now 6
counter.reset()
// the counter's value is now 0
```

Function parameters can have both a name (for use within the function's body) and an argument label (for use when calling the function), as described in Function Argument Labels and Parameter Names. The same is true for method parameters, because methods are just functions that are associated with a type.

The self Property

Every instance of a type has an implicit property called `self`, which is exactly equivalent to the instance itself. You use the `self` property to refer to the current instance within its own instance methods.

The `increment()` method in the example above could have been written like this:

```
func increment() {
    self.count += 1
}
```

In practice, you don't need to write `self` in your code very often. If you don't explicitly write `self`, Swift assumes that you are referring to a property or method of the current instance whenever you use a known

property or method name within a method. This assumption is demonstrated by the use of `count` (rather than `self.count`) inside the three instance methods for `Counter`.

The main exception to this rule occurs when a parameter name for an instance method has the same name as a property of that instance. In this situation, the parameter name takes precedence, and it becomes necessary to refer to the property in a more qualified way. You use the `self` property to distinguish between the parameter name and the property name.

Here, `self` disambiguates between a method parameter called `x` and an instance property that's also called `x`:

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOf(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOf(x: 1.0) {
    print("This point is to the right of the line where x == 1.0")
}
// Prints "This point is to the right of the line where x == 1.0"
```

Without the `self` prefix, Swift would assume that both uses of `x` referred to the method parameter called `x`.

Modifying Value Types from Within Instance Methods
Structures and enumerations are *value types*. By default, the properties of a value type can't be modified from within its instance methods.

However, if you need to modify the properties of your structure or enumeration within a particular method, you can opt in to *mutating* behavior for that method. The method can then mutate (that is, change) its properties from within the method, and any changes that it makes are written back to the original structure when the method ends. The method can also assign a completely new instance to its implicit `self` property, and this new instance will replace the existing one when the method ends.

You can opt in to this behavior by placing the `mutating` keyword before the `func` keyword for that method:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveBy(x: 2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")
// Prints "The point is now at (3.0, 4.0)"
```

The `Point` structure above defines a mutating `moveBy(x:y:)` method, which moves a `Point` instance by a certain amount. Instead of returning a

new point, this method actually modifies the point on which it's called. The `mutating` keyword is added to its definition to enable it to modify its properties.

Note that you can't call a mutating method on a constant of structure type, because its properties can't be changed, even if they're variable properties, as described in [Stored Properties of Constant Structure Instances]:

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveBy(x: 2.0, y: 3.0)
// this will report an error
```

Assigning to self Within a Mutating Method

Mutating methods can assign an entirely new instance to the implicit `self` property. The `Point` example shown above could have been written in the following way instead:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY:
     Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

This version of the mutating `moveBy(x:y:)` method creates a new structure whose `x` and `y` values are set to the target location. The end result of calling this alternative version of the method will be exactly the same as for calling the earlier version.

Mutating methods for enumerations can set the implicit `self` parameter to be a different case from the same enumeration:

```
enum TriStateSwitch {
    case off, low, high
    mutating func next() {
        switch self {
        case .off:
            self = .low
        case .low:
            self = .high
        case .high:
            self = .off
        }
    }
}
var ovenLight = TriStateSwitch.low
ovenLight.next()
// ovenLight is now equal to .high
ovenLight.next()
// ovenLight is now equal to .off
```

This example defines an enumeration for a three-state switch. The switch cycles between three different power states (`off`, `low` and `high`) every time its `next()` method is called.

Type Methods

Instance methods, as described above, are methods that you call on an instance of a particular type. You can also define methods that are called on the type itself. These kinds of methods are called *type methods*. You indicate type methods by writing the `static` keyword before the method's `func` keyword. Classes can use the `class` keyword instead, to allow subclasses to override the superclass's implementation of that method.

> NOTE
>
> In Objective-C, you can define type-level methods only for Objective-C classes. In Swift, you can define type-level methods for all classes, structures, and enumerations. Each type method is explicitly scoped to the type it supports.

Type methods are called with dot syntax, like instance methods. However, you call type methods on the type, not on an instance of that type. Here's how you call a type method on a class called `SomeClass`:

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
SomeClass.someTypeMethod()
```

Within the body of a type method, the implicit `self` property refers to the type itself, rather than an instance of that type. This means that you can use `self` to disambiguate between type properties and type method parameters, just as you do for instance properties and instance method parameters.

More generally, any unqualified method and property names that you use within the body of a type method will refer to other type-level methods and properties. A type method can call another type method with the other

method's name, without needing to prefix it with the type name. Similarly, type methods on structures and enumerations can access type properties by using the type property's name without a type name prefix.

The example below defines a structure called `LevelTracker`, which tracks a player's progress through the different levels or stages of a game. It's a single-player game, but can store information for multiple players on a single device.

All of the game's levels (apart from level one) are locked when the game is first played. Every time a player finishes a level, that level is unlocked for all players on the device. The `LevelTracker` structure uses type properties and methods to keep track of which levels of the game have been unlocked. It also tracks the current level for an individual player.

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    var currentLevel = 1

    static func unlock(_ level: Int) {
        if level > highestUnlockedLevel {
 highestUnlockedLevel = level }
    }

    static func isUnlocked(_ level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }

    @discardableResult
    mutating func advance(to level: Int) -> Bool {
        if LevelTracker.isUnlocked(level) {
            currentLevel = level
            return true
        } else {
            return false
        }
    }
}
```

The `LevelTracker` structure keeps track of the highest level that any player has unlocked. This value is stored in a type property called `highestUnlockedLevel`.

`LevelTracker` also defines two type functions to work with the `highestUnlockedLevel` property. The first is a type function called `unlock(_:)`, which updates the value of `highestUnlockedLevel` whenever a new level is unlocked. The second is a convenience type function called `isUnlocked(_:)`, which returns `true` if a particular level number is already unlocked. (Note that these type methods can access the `highestUnlockedLevel` type property without your needing to write it as `LevelTracker.highestUnlockedLevel`.)

In addition to its type property and type methods, `LevelTracker` tracks an individual player's progress through the game. It uses an instance property called `currentLevel` to track the level that a player is currently playing.

To help manage the `currentLevel` property, `LevelTracker` defines an instance method called `advance(to:)`. Before updating `currentLevel`, this method checks whether the requested new level is already unlocked. The `advance(to:)` method returns a Boolean value to indicate whether or not it was actually able to set `currentLevel`. Because it's not necessarily a mistake for code that calls the `advance(to:)` method to ignore the return value, this function is marked with the `@discardableResult` attribute. For more information about this attribute, see Attributes.

The `LevelTracker` structure is used with the `Player` class, shown below, to track and update the progress of an individual player:

```
class Player {
    var tracker = LevelTracker()
    let playerName: String
    func complete(level: Int) {
        LevelTracker.unlock(level + 1)
        tracker.advance(to: level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

The `Player` class creates a new instance of `LevelTracker` to track that player's progress. It also provides a method called `complete(level:)`, which is called whenever a player completes a particular level. This method unlocks the next level for all players and updates the player's progress to move them to the next level. (The Boolean return value of `advance(to:)` is ignored, because the level is known to have been unlocked by the call to `LevelTracker.unlock(_:)` on the previous line.)

You can create an instance of the `Player` class for a new player, and see what happens when the player completes level one:

```
var player = Player(name: "Argyrios")
player.complete(level: 1)
print("highest unlocked level is now \
     (LevelTracker.highestUnlockedLevel)")
// Prints "highest unlocked level is now 2"
```

If you create a second player, whom you try to move to a level that's not yet unlocked by any player in the game, the attempt to set the player's current level fails:

```
player = Player(name: "Beto")
if player.tracker.advance(to: 6) {
    print("player is now on level 6")
} else {
    print("level 6 hasn't yet been unlocked")
}
// Prints "level 6 hasn't yet been unlocked"
```

Subscripts

Classes, structures, and enumerations can define *subscripts*, which are shortcuts for accessing the member elements of a collection, list, or sequence. You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval. For example, you access elements in an `Array` instance as `someArray[index]` and elements in a `Dictionary` instance as `someDictionary[key]`.

You can define multiple subscripts for a single type, and the appropriate subscript overload to use is selected based on the type of index value you pass to the subscript. Subscripts aren't limited to a single dimension, and you can define subscripts with multiple input parameters to suit your custom type's needs.

Subscript Syntax

Subscripts enable you to query instances of a type by writing one or more values in square brackets after the instance name. Their syntax is similar to both instance method syntax and computed property syntax. You write subscript definitions with the `subscript` keyword, and specify one or more input parameters and a return type, in the same way as instance methods. Unlike instance methods, subscripts can be read-write or read-only. This behavior is communicated by a getter and setter in the same way as for computed properties:

```
subscript(index: Int) -> Int {
    get {
        // Return an appropriate subscript value here.
    }
    set(newValue) {
        // Perform a suitable setting action here.
    }
}
```

The type of `newValue` is the same as the return value of the subscript. As with computed properties, you can choose not to specify the setter's `(newValue)` parameter. A default parameter called `newValue` is provided to your setter if you don't provide one yourself.

As with read-only computed properties, you can simplify the declaration of a read-only subscript by removing the `get` keyword and its braces:

```
subscript(index: Int) -> Int {
    // Return an appropriate subscript value here.
}
```

Here's an example of a read-only subscript implementation, which defines a `TimesTable` structure to represent an *n*-times-table of integers:

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
print("six times three is \(threeTimesTable[6])")
// Prints "six times three is 18"
```

In this example, a new instance of `TimesTable` is created to represent the three-times-table. This is indicated by passing a value of `3` to the structure's `initializer` as the value to use for the instance's `multiplier` parameter.

You can query the `threeTimesTable` instance by calling its subscript, as shown in the call to `threeTimesTable[6]`. This requests the sixth entry in the three-times-table, which returns a value of `18`, or `3` times `6`.

> NOTE
>
> An *n*-times-table is based on a fixed mathematical rule. It isn't appropriate to set `threeTimesTable[someIndex]` to a new value, and so the subscript for `TimesTable` is defined as a read-only subscript.

Subscript Usage

The exact meaning of "subscript" depends on the context in which it's used. Subscripts are typically used as a shortcut for accessing the member elements in a collection, list, or sequence. You are free to implement

subscripts in the most appropriate way for your particular class or structure's functionality.

For example, Swift's `Dictionary` type implements a subscript to set and retrieve the values stored in a `Dictionary` instance. You can set a value in a dictionary by providing a key of the dictionary's key type within subscript brackets, and assigning a value of the dictionary's value type to the subscript:

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

The example above defines a variable called `numberOfLegs` and initializes it with a dictionary literal containing three key-value pairs. The type of the `numberOfLegs` dictionary is inferred to be `[String: Int]`. After creating the dictionary, this example uses subscript assignment to add a `String` key of `"bird"` and an `Int` value of `2` to the dictionary.

For more information about `Dictionary` subscripting, see [Accessing and Modifying a Dictionary](#).

> NOTE
>
> Swift's `Dictionary` type implements its key-value subscripting as a subscript that takes and returns an *optional* type. For the `numberOfLegs` dictionary above, the key-value subscript takes and returns a value of type `Int?`, or "optional int". The `Dictionary` type uses an optional subscript type to model the fact that not every key will have a value, and to give a way to delete a value for a key by assigning a `nil` value for that key.

Subscript Options

Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return a value of any type.

Like functions, subscripts can take a varying number of parameters and provide default values for their parameters, as discussed in Variadic Parameters and Default Parameter Values. However, unlike functions, subscripts can't use in-out parameters.

A class or structure can provide as many subscript implementations as it needs, and the appropriate subscript to be used will be inferred based on the types of the value or values that are contained within the subscript brackets at the point that the subscript is used. This definition of multiple subscripts is known as *subscript overloading*.

While it's most common for a subscript to take a single parameter, you can also define a subscript with multiple parameters if it's appropriate for your type. The following example defines a `Matrix` structure, which represents a two-dimensional matrix of `Double` values. The `Matrix` structure's subscript takes two integer parameters:

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
        }
```

```
        }
    }
```

`Matrix` provides an initializer that takes two parameters called `rows` and `columns`, and creates an array that's large enough to store `rows * columns` values of type `Double`. Each position in the matrix is given an initial value of `0.0`. To achieve this, the array's size, and an initial cell value of `0.0`, are passed to an array initializer that creates and initializes a new array of the correct size. This initializer is described in more detail in [Creating an Array with a Default Value](#).

You can construct a new `Matrix` instance by passing an appropriate row and column count to its initializer:

```
var matrix = Matrix(rows: 2, columns: 2)
```

The example above creates a new `Matrix` instance with two rows and two columns. The `grid` array for this `Matrix` instance is effectively a flattened version of the matrix, as read from top left to bottom right:

```
grid = [0.0, 0.0, 0.0, 0.0]

              column
             0      1
row  0  [ [ 0.0, 0.0,
     1    0.0, 0.0 ] ]
```

Values in the matrix can be set by passing row and column values into the subscript, separated by a comma:

```
matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

These two statements call the subscript's setter to set a value of `1.5` in the top right position of the matrix (where `row` is `0` and `column` is `1`), and `3.2` in the bottom left position (where `row` is `1` and `column` is `0`):

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

The `Matrix` subscript's getter and setter both contain an assertion to check that the subscript's `row` and `column` values are valid. To assist with these assertions, `Matrix` includes a convenience method called `indexIsValid(row:column:)`, which checks whether the requested `row` and `column` are inside the bounds of the matrix:

```
func indexIsValid(row: Int, column: Int) -> Bool {
    return row >= 0 && row < rows && column >= 0 &&
     column < columns
}
```

An assertion is triggered if you try to access a subscript that's outside of the matrix bounds:

```
let someValue = matrix[2, 2]
// This triggers an assert, because [2, 2] is
     outside of the matrix bounds.
```

Type Subscripts

Instance subscripts, as described above, are subscripts that you call on an instance of a particular type. You can also define subscripts that are called

on the type itself. This kind of subscript is called a *type subscript*. You indicate a type subscript by writing the `static` keyword before the `subscript` keyword. Classes can use the `class` keyword instead, to allow subclasses to override the superclass's implementation of that subscript. The example below shows how you define and call a type subscript:

```
enum Planet: Int {
    case mercury = 1, venus, earth, mars, jupiter,
     saturn, uranus, neptune
    static subscript(n: Int) -> Planet {
        return Planet(rawValue: n)!
    }
}
let mars = Planet[4]
print(mars)
```

Inheritance

A class can *inherit* methods, properties, and other characteristics from another class. When one class inherits from another, the inheriting class is known as a *subclass*, and the class it inherits from is known as its *superclass*. Inheritance is a fundamental behavior that differentiates classes from other types in Swift.

Classes in Swift can call and access methods, properties, and subscripts belonging to their superclass and can provide their own overriding versions of those methods, properties, and subscripts to refine or modify their behavior. Swift helps to ensure your overrides are correct by checking that the override definition has a matching superclass definition.

Classes can also add property observers to inherited properties in order to be notified when the value of a property changes. Property observers can be added to any property, regardless of whether it was originally defined as a stored or computed property.

Defining a Base Class

Any class that doesn't inherit from another class is known as a *base class*.

> NOTE
>
> Swift classes don't inherit from a universal base class. Classes you define without specifying a superclass automatically become base classes for you to build upon.

The example below defines a base class called `Vehicle`. This base class defines a stored property called `currentSpeed`, with a default value of `0.0` (inferring a property type of `Double`). The `currentSpeed` property's value

is used by a read-only computed `String` property called `description` to create a description of the vehicle.

The `Vehicle` base class also defines a method called `makeNoise`. This method doesn't actually do anything for a base `Vehicle` instance, but will be customized by subclasses of `Vehicle` later on:

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // do nothing - an arbitrary vehicle doesn't necessarily make a noise
    }
}
```

You create a new instance of `Vehicle` with *initializer syntax*, which is written as a type name followed by empty parentheses:

```
let someVehicle = Vehicle()
```

Having created a new `Vehicle` instance, you can access its `description` property to print a human-readable description of the vehicle's current speed:

```
print("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

The `Vehicle` class defines common characteristics for an arbitrary vehicle, but isn't much use in itself. To make it more useful, you need to refine it to describe more specific kinds of vehicles.

Subclassing

Subclassing is the act of basing a new class on an existing class. The subclass inherits characteristics from the existing class, which you can then refine. You can also add new characteristics to the subclass.

To indicate that a subclass has a superclass, write the subclass name before the superclass name, separated by a colon:

```
class SomeSubclass: SomeSuperclass {
    // subclass definition goes here
}
```

The following example defines a subclass called `Bicycle`, with a superclass of `Vehicle`:

```
class Bicycle: Vehicle {
    var hasBasket = false
}
```

The new `Bicycle` class automatically gains all of the characteristics of `Vehicle`, such as its `currentSpeed` and `description` properties and its `makeNoise()` method.

In addition to the characteristics it inherits, the `Bicycle` class defines a new stored property, `hasBasket`, with a default value of `false` (inferring a type

of `Bool` for the property).

By default, any new `Bicycle` instance you create will not have a basket. You can set the `hasBasket` property to `true` for a particular `Bicycle` instance after that instance is created:

```
let bicycle = Bicycle()
bicycle.hasBasket = true
```

You can also modify the inherited `currentSpeed` property of a `Bicycle` instance, and query the instance's inherited `description` property:

```
bicycle.currentSpeed = 15.0
print("Bicycle: \(bicycle.description)")
// Bicycle: traveling at 15.0 miles per hour
```

Subclasses can themselves be subclassed. The next example creates a subclass of `Bicycle` for a two-seater bicycle known as a "tandem":

```
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
```

`Tandem` inherits all of the properties and methods from `Bicycle`, which in turn inherits all of the properties and methods from `Vehicle`. The `Tandem` subclass also adds a new stored property called `currentNumberOfPassengers`, with a default value of `0`.

If you create an instance of `Tandem`, you can work with any of its new and inherited properties, and query the read-only `description` property it inherits from `Vehicle`:

```
let tandem = Tandem()
tandem.hasBasket = true
tandem.currentNumberOfPassengers = 2
tandem.currentSpeed = 22.0
print("Tandem: \(tandem.description)")
// Tandem: traveling at 22.0 miles per hour
```

Overriding

A subclass can provide its own custom implementation of an instance method, type method, instance property, type property, or subscript that it would otherwise inherit from a superclass. This is known as *overriding*.

To override a characteristic that would otherwise be inherited, you prefix your overriding definition with the `override` keyword. Doing so clarifies that you intend to provide an override and haven't provided a matching definition by mistake. Overriding by accident can cause unexpected behavior, and any overrides without the `override` keyword are diagnosed as an error when your code is compiled.

The `override` keyword also prompts the Swift compiler to check that your overriding class's superclass (or one of its parents) has a declaration that matches the one you provided for the override. This check ensures that your overriding definition is correct.

Accessing Superclass Methods, Properties, and Subscripts

When you provide a method, property, or subscript override for a subclass, it's sometimes useful to use the existing superclass implementation as part

of your override. For example, you can refine the behavior of that existing implementation, or store a modified value in an existing inherited variable.

Where this is appropriate, you access the superclass version of a method, property, or subscript by using the `super` prefix:

- An overridden method named `someMethod()` can call the superclass version of `someMethod()` by calling `super.someMethod()` within the overriding method implementation.
- An overridden property called `someProperty` can access the superclass version of `someProperty` as `super.someProperty` within the overriding getter or setter implementation.
- An overridden subscript for `someIndex` can access the superclass version of the same subscript as `super[someIndex]` from within the overriding subscript implementation.

Overriding Methods

You can override an inherited instance or type method to provide a tailored or alternative implementation of the method within your subclass.

The following example defines a new subclass of `Vehicle` called `Train`, which overrides the `makeNoise()` method that `Train` inherits from `Vehicle`:

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}
```

If you create a new instance of `Train` and call its `makeNoise()` method, you can see that the `Train` subclass version of the method is called:

```
let train = Train()
train.makeNoise()
// Prints "Choo Choo"
```

Overriding Properties

You can override an inherited instance or type property to provide your own custom getter and setter for that property, or to add property observers to enable the overriding property to observe when the underlying property value changes.

Overriding Property Getters and Setters

You can provide a custom getter (and setter, if appropriate) to override *any* inherited property, regardless of whether the inherited property is implemented as a stored or computed property at source. The stored or computed nature of an inherited property isn't known by a subclass—it only knows that the inherited property has a certain name and type. You must always state both the name and the type of the property you are overriding, to enable the compiler to check that your override matches a superclass property with the same name and type.

You can present an inherited read-only property as a read-write property by providing both a getter and a setter in your subclass property override. You can't, however, present an inherited read-write property as a read-only property.

> NOTE
>
> If you provide a setter as part of a property override, you must also provide a getter for that override. If you don't want to modify the inherited property's value within the overriding getter, you can simply pass through the inherited value by returning `super.someProperty` from the getter, where `someProperty` is the name of the property you are overriding.

The following example defines a new class called `Car`, which is a subclass of `Vehicle`. The `Car` class introduces a new stored property called `gear`, with a default integer value of `1`. The `Car` class also overrides the `description` property it inherits from `Vehicle`, to provide a custom description that includes the current gear:

```
class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \
 (gear)"
    }
}
```

The override of the `description` property starts by calling `super.description`, which returns the `Vehicle` class's `description` property. The `Car` class's version of `description` then adds some extra text onto the end of this description to provide information about the current gear.

If you create an instance of the `Car` class and set its `gear` and `currentSpeed` properties, you can see that its `description` property returns the tailored description defined within the `Car` class:

```
let car = Car()
car.currentSpeed = 25.0
car.gear = 3
print("Car: \(car.description)")
// Car: traveling at 25.0 miles per hour in gear 3
```

Overriding Property Observers

You can use property overriding to add property observers to an inherited property. This enables you to be notified when the value of an inherited property changes, regardless of how that property was originally implemented. For more information on property observers, see [Property Observers](#).

> NOTE
>
> You can't add property observers to inherited constant stored properties or inherited read-only computed properties. The value of these properties can't be set, and so it isn't appropriate to provide a `willSet` or `didSet` implementation as part of an override.
>
> Note also that you can't provide both an overriding setter and an overriding property observer for the same property. If you want to observe changes to a property's value, and you are already providing a custom setter for that property, you can simply observe any value changes from within the custom setter.

The following example defines a new class called `AutomaticCar`, which is a subclass of `Car`. The `AutomaticCar` class represents a car with an automatic gearbox, which automatically selects an appropriate gear to use based on the current speed:

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

Whenever you set the `currentSpeed` property of an `AutomaticCar` instance, the property's `didSet` observer sets the instance's `gear` property to an appropriate choice of gear for the new speed. Specifically, the property observer chooses a gear that's the new `currentSpeed` value divided by `10`, rounded down to the nearest integer, plus `1`. A speed of `35.0` produces a gear of `4`:

```
let automatic = AutomaticCar()
automatic.currentSpeed = 35.0
print("AutomaticCar: \(automatic.description)")
// AutomaticCar: traveling at 35.0 miles per hour in
     gear 4
```

Preventing Overrides

You can prevent a method, property, or subscript from being overridden by marking it as *final*. Do this by writing the `final` modifier before the method, property, or subscript's introducer keyword (such as `final var`, `final func`, `final class func`, and `final subscript`).

Any attempt to override a final method, property, or subscript in a subclass is reported as a compile-time error. Methods, properties, or subscripts that you add to a class in an extension can also be marked as final within the extension's definition.

You can mark an entire class as final by writing the `final` modifier before the `class` keyword in its class definition (`final class`). Any attempt to subclass a final class is reported as a compile-time error.

Initialization

Initialization is the process of preparing an instance of a class, structure, or enumeration for use. This process involves setting an initial value for each stored property on that instance and performing any other setup or initialization that's required before the new instance is ready for use.

You implement this initialization process by defining *initializers*, which are like special methods that can be called to create a new instance of a particular type. Unlike Objective-C initializers, Swift initializers don't return a value. Their primary role is to ensure that new instances of a type are correctly initialized before they're used for the first time.

Instances of class types can also implement a *deinitializer*, which performs any custom cleanup just before an instance of that class is deallocated. For more information about deinitializers, see Deinitialization.

Setting Initial Values for Stored Properties

Classes and structures *must* set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created. Stored properties can't be left in an indeterminate state.

You can set an initial value for a stored property within an initializer, or by assigning a default property value as part of the property's definition. These actions are described in the following sections.

> NOTE
>
> When you assign a default value to a stored property, or set its initial value within an initializer, the value of that property is set directly, without calling any property observers.

Initializers

Initializers are called to create a new instance of a particular type. In its simplest form, an initializer is like an instance method with no parameters, written using the `init` keyword:

```
init() {
    // perform some initialization here
}
```

The example below defines a new structure called `Fahrenheit` to store temperatures expressed in the Fahrenheit scale. The `Fahrenheit` structure has one stored property, `temperature`, which is of type `Double`:

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// Prints "The default temperature is 32.0° Fahrenheit"
```

The structure defines a single initializer, `init`, with no parameters, which initializes the stored temperature with a value of `32.0` (the freezing point of water in degrees Fahrenheit).

Default Property Values

You can set the initial value of a stored property from within an initializer, as shown above. Alternatively, specify a *default property value* as part of the property's declaration. You specify a default property value by assigning an initial value to the property when it's defined.

> NOTE
>
> If a property always takes the same initial value, provide a default value rather than setting a value within an initializer. The end result is the same, but the default value ties the property's initialization more closely to its declaration. It makes for shorter, clearer initializers and enables you to infer the type of the property from its default value. The default value also makes it easier for you to take advantage of default initializers and initializer inheritance, as described later in this chapter.

You can write the `Fahrenheit` structure from above in a simpler form by providing a default value for its `temperature` property at the point that the property is declared:

```
struct Fahrenheit {
    var temperature = 32.0
}
```

Customizing Initialization

You can customize the initialization process with input parameters and optional property types, or by assigning constant properties during initialization, as described in the following sections.

Initialization Parameters

You can provide *initialization parameters* as part of an initializer's definition, to define the types and names of values that customize the initialization process. Initialization parameters have the same capabilities and syntax as function and method parameters.

The following example defines a structure called `Celsius`, which stores temperatures expressed in degrees Celsius. The `Celsius` structure implements two custom initializers called `init(fromFahrenheit:)` and `init(fromKelvin:)`, which initialize a new instance of the structure with a value from a different temperature scale:

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

The first initializer has a single initialization parameter with an argument label of `fromFahrenheit` and a parameter name of `fahrenheit`. The

second initializer has a single initialization parameter with an argument label of `fromKelvin` and a parameter name of `kelvin`. Both initializers convert their single argument into the corresponding Celsius value and store this value in a property called `temperatureInCelsius`.

Parameter Names and Argument Labels

As with function and method parameters, initialization parameters can have both a parameter name for use within the initializer's body and an argument label for use when calling the initializer.

However, initializers don't have an identifying function name before their parentheses in the way that functions and methods do. Therefore, the names and types of an initializer's parameters play a particularly important role in identifying which initializer should be called. Because of this, Swift provides an automatic argument label for *every* parameter in an initializer if you don't provide one.

The following example defines a structure called `Color`, with three constant properties called `red`, `green`, and `blue`. These properties store a value between `0.0` and `1.0` to indicate the amount of red, green, and blue in the color.

`Color` provides an initializer with three appropriately named parameters of type `Double` for its red, green, and blue components. `Color` also provides a second initializer with a single `white` parameter, which is used to provide the same value for all three color components.

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
    init(white: Double) {
        red   = white
        green = white
        blue  = white
    }
}
```

Both initializers can be used to create a new `Color` instance, by providing named values for each initializer parameter:

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
let halfGray = Color(white: 0.5)
```

Note that it isn't possible to call these initializers without using argument labels. Argument labels must always be used in an initializer if they're defined, and omitting them is a compile-time error:

```
let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - argument
     labels are required
```

Initializer Parameters Without Argument Labels
If you don't want to use an argument label for an initializer parameter, write an underscore (_) instead of an explicit argument label for that parameter to override the default behavior.

Here's an expanded version of the `Celsius` example from [Initialization Parameters](#) above, with an additional initializer to create a new `Celsius` instance from a `Double` value that's already in the Celsius scale:

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
    init(_ celsius: Double) {
        temperatureInCelsius = celsius
    }
}
let bodyTemperature = Celsius(37.0)
// bodyTemperature.temperatureInCelsius is 37.0
```

The initializer call `Celsius(37.0)` is clear in its intent without the need for an argument label. It's therefore appropriate to write this initializer as `init(_ celsius: Double)` so that it can be called by providing an unnamed `Double` value.

Optional Property Types
If your custom type has a stored property that's logically allowed to have "no value"—perhaps because its value can't be set during initialization, or because it's allowed to have "no value" at some later point—declare the property with an *optional* type. Properties of optional type are automatically initialized with a value of `nil`, indicating that the property is deliberately intended to have "no value yet" during initialization.

The following example defines a class called `SurveyQuestion`, with an optional `String` property called `response`:

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}
let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")
cheeseQuestion.ask()
// Prints "Do you like cheese?"
cheeseQuestion.response = "Yes, I do like cheese."
```

The response to a survey question can't be known until it's asked, and so the `response` property is declared with a type of `String?`, or "optional

`String`". It's automatically assigned a default value of `nil`, meaning "no string yet", when a new instance of `SurveyQuestion` is initialized.

Assigning Constant Properties During Initialization

You can assign a value to a constant property at any point during initialization, as long as it's set to a definite value by the time initialization finishes. Once a constant property is assigned a value, it can't be further modified.

> NOTE
>
> For class instances, a constant property can be modified during initialization only by the class that introduces it. It can't be modified by a subclass.

You can revise the `SurveyQuestion` example from above to use a constant property rather than a variable property for the `text` property of the question, to indicate that the question doesn't change once an instance of `SurveyQuestion` is created. Even though the `text` property is now a constant, it can still be set within the class's initializer:

```swift
class SurveyQuestion {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}
let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
// Prints "How about beets?"
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

Default Initializers

Swift provides a *default initializer* for any structure or class that provides default values for all of its properties and doesn't provide at least one initializer itself. The default initializer simply creates a new instance with all of its properties set to their default values.

This example defines a class called `ShoppingListItem`, which encapsulates the name, quantity, and purchase state of an item in a shopping list:

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}
var item = ShoppingListItem()
```

Because all properties of the `ShoppingListItem` class have default values, and because it's a base class with no superclass, `ShoppingListItem` automatically gains a default initializer implementation that creates a new instance with all of its properties set to their default values. (The `name` property is an optional `String` property, and so it automatically receives a default value of `nil`, even though this value isn't written in the code.) The example above uses the default initializer for the `ShoppingListItem` class to create a new instance of the class with initializer syntax, written as `ShoppingListItem()`, and assigns this new instance to a variable called `item`.

Memberwise Initializers for Structure Types

Structure types automatically receive a *memberwise initializer* if they don't define any of their own custom initializers. Unlike a default initializer, the structure receives a memberwise initializer even if it has stored properties that don't have default values.

The memberwise initializer is a shorthand way to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name.

The example below defines a structure called `Size` with two properties called `width` and `height`. Both properties are inferred to be of type `Double` by assigning a default value of `0.0`.

The `Size` structure automatically receives an `init(width:height:)` memberwise initializer, which you can use to initialize a new `Size` instance:

```
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

When you call a memberwise initializer, you can omit values for any properties that have default values. In the example above, the `Size` structure has a default value for both its `height` and `width` properties. You can omit either property or both properties, and the initializer uses the default value for anything you omit—for example:

```
let zeroByTwo = Size(height: 2.0)
print(zeroByTwo.width, zeroByTwo.height)
// Prints "0.0 2.0"

let zeroByZero = Size()
print(zeroByZero.width, zeroByZero.height)
// Prints "0.0 0.0"
```

Initializer Delegation for Value Types

Initializers can call other initializers to perform part of an instance's initialization. This process, known as *initializer delegation*, avoids duplicating code across multiple initializers.

The rules for how initializer delegation works, and for what forms of delegation are allowed, are different for value types and class types. Value types (structures and enumerations) don't support inheritance, and so their initializer delegation process is relatively simple, because they can only delegate to another initializer that they provide themselves. Classes, however, can inherit from other classes, as described in Inheritance. This means that classes have additional responsibilities for ensuring that all stored properties they inherit are assigned a suitable value during initialization. These responsibilities are described in Class Inheritance and Initialization below.

For value types, you use `self.init` to refer to other initializers from the same value type when writing your own custom initializers. You can call `self.init` only from within an initializer.

Note that if you define a custom initializer for a value type, you will no longer have access to the default initializer (or the memberwise initializer, if it's a structure) for that type. This constraint prevents a situation in which additional essential setup provided in a more complex initializer is accidentally circumvented by someone using one of the automatic initializers.

> NOTE
>
> If you want your custom value type to be initializable with the default initializer and memberwise initializer, and also with your own custom initializers, write your custom initializers in an extension rather than as part of the value type's original implementation. For more information, see Extensions.

The following example defines a custom `Rect` structure to represent a geometric rectangle. The example requires two supporting structures called `Size` and `Point`, both of which provide default values of `0.0` for all of their properties:

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
```

You can initialize the `Rect` structure below in one of three ways—by using its default zero-initialized `origin` and `size` property values, by providing a specific origin point and size, or by providing a specific center point and size. These initialization options are represented by three custom initializers that are part of the `Rect` structure's definition:

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

The first `Rect` initializer, `init()`, is functionally the same as the default initializer that the structure would have received if it didn't have its own custom initializers. This initializer has an empty body, represented by an empty pair of curly braces `{}`. Calling this initializer returns a `Rect` instance whose `origin` and `size` properties are both initialized with the default values of `Point(x: 0.0, y: 0.0)` and `Size(width: 0.0, height: 0.0)` from their property definitions:

```
let basicRect = Rect()
// basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)
```

The second `Rect` initializer, `init(origin:size:)`, is functionally the same as the memberwise initializer that the structure would have received if it didn't have its own custom initializers. This initializer simply assigns the `origin` and `size` argument values to the appropriate stored properties:

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
                      size: Size(width: 5.0, height: 5.0))
// originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)
```

The third `Rect` initializer, `init(center:size:)`, is slightly more complex. It starts by calculating an appropriate origin point based on a `center` point and a `size` value. It then calls (or *delegates*) to the `init(origin:size:)` initializer, which stores the new origin and size values in the appropriate properties:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

The `init(center:size:)` initializer could have assigned the new values of `origin` and `size` to the appropriate properties itself. However, it's more convenient (and clearer in intent) for the `init(center:size:)` initializer to take advantage of an existing initializer that already provides exactly that functionality.

> NOTE
>
> For an alternative way to write this example without defining the `init()` and `init(origin:size:)` initializers yourself, see Extensions.

Class Inheritance and Initialization

All of a class's stored properties—including any properties the class inherits from its superclass—*must* be assigned an initial value during initialization.

Swift defines two kinds of initializers for class types to help ensure all stored properties receive an initial value. These are known as designated initializers and convenience initializers.

Designated Initializers and Convenience Initializers

Designated initializers are the primary initializers for a class. A designated initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain.

Classes tend to have very few designated initializers, and it's quite common for a class to have only one. Designated initializers are "funnel" points through which initialization takes place, and through which the initialization process continues up the superclass chain.

Every class must have at least one designated initializer. In some cases, this requirement is satisfied by inheriting one or more designated initializers from a superclass, as described in Automatic Initializer Inheritance below.

Convenience initializers are secondary, supporting initializers for a class. You can define a convenience initializer to call a designated initializer from the same class as the convenience initializer with some of the designated

initializer's parameters set to default values. You can also define a convenience initializer to create an instance of that class for a specific use case or input value type.

You don't have to provide convenience initializers if your class doesn't require them. Create convenience initializers whenever a shortcut to a common initialization pattern will save time or make initialization of the class clearer in intent.

Syntax for Designated and Convenience Initializers
Designated initializers for classes are written in the same way as simple initializers for value types:

```
init(parameters) {
    statements
}
```

Convenience initializers are written in the same style, but with the `convenience` modifier placed before the `init` keyword, separated by a space:

```
convenience init(parameters) {
    statements
}
```

Initializer Delegation for Class Types
To simplify the relationships between designated and convenience initializers, Swift applies the following three rules for delegation calls between initializers:

Rule 1

A designated initializer must call a designated initializer from its immediate superclass.

Rule 2

A convenience initializer must call another initializer from the *same* class.

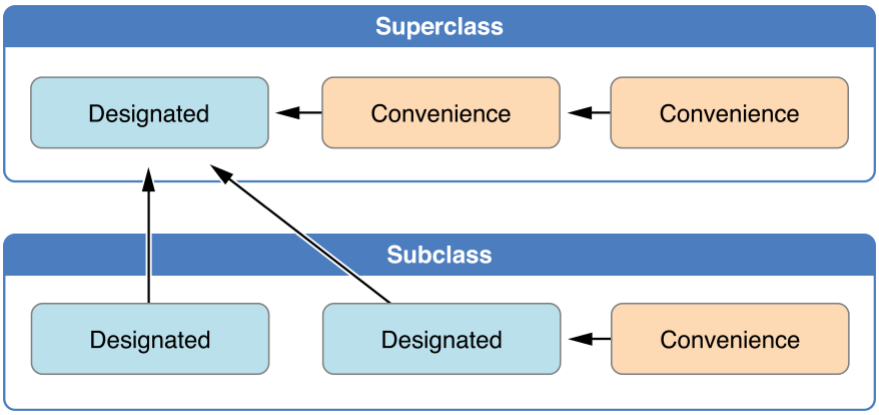
Rule 3

A convenience initializer must ultimately call a designated initializer.

A simple way to remember this is:

- Designated initializers must always delegate *up*.
- Convenience initializers must always delegate *across*.

These rules are illustrated in the figure below:

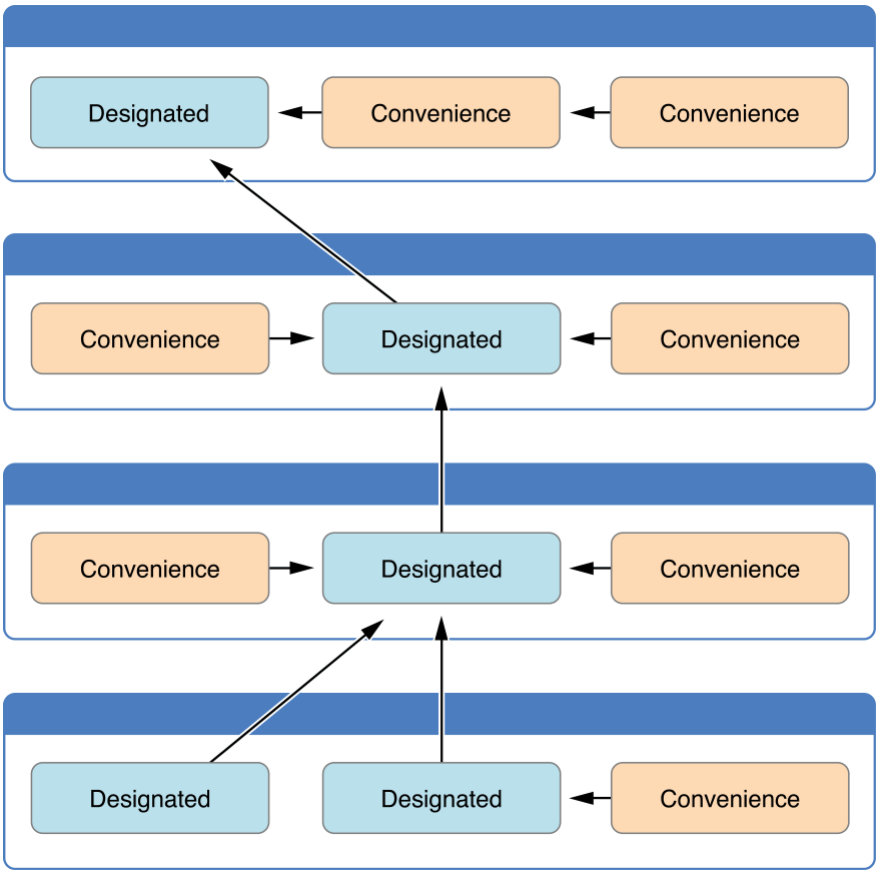


Here, the superclass has a single designated initializer and two convenience initializers. One convenience initializer calls another convenience initializer, which in turn calls the single designated initializer. This satisfies rules 2 and 3 from above. The superclass doesn't itself have a further superclass, and so rule 1 doesn't apply.

The subclass in this figure has two designated initializers and one convenience initializer. The convenience initializer must call one of the two designated initializers, because it can only call another initializer from the same class. This satisfies rules 2 and 3 from above. Both designated initializers must call the single designated initializer from the superclass, to satisfy rule 1 from above.

> NOTE
>
> These rules don't affect how users of your classes *create* instances of each class. Any initializer in the diagram above can be used to create a fully initialized instance of the class they belong to. The rules only affect how you write the implementation of the class's initializers.

The figure below shows a more complex class hierarchy for four classes. It illustrates how the designated initializers in this hierarchy act as "funnel" points for class initialization, simplifying the interrelationships among classes in the chain:



Two-Phase Initialization

Class initialization in Swift is a two-phase process. In the first phase, each stored property is assigned an initial value by the class that introduced it. Once the initial state for every stored property has been determined, the second phase begins, and each class is given the opportunity to customize its stored properties further before the new instance is considered ready for use.

The use of a two-phase initialization process makes initialization safe, while still giving complete flexibility to each class in a class hierarchy. Two-phase initialization prevents property values from being accessed before they're initialized, and prevents property values from being set to a different value by another initializer unexpectedly.

> NOTE
>
> Swift's two-phase initialization process is similar to initialization in Objective-C. The main difference is that during phase 1, Objective-C assigns zero or null values (such as `0` or `nil`) to every property. Swift's initialization flow is more flexible in that it lets you set custom initial values, and can cope with types for which `0` or `nil` isn't a valid default value.

Swift's compiler performs four helpful safety-checks to make sure that two-phase initialization is completed without error:

Safety check 1

> A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer.

As mentioned above, the memory for an object is only considered fully initialized once the initial state of all of its stored properties is known. In order for this rule to be satisfied, a designated initializer must make sure that all of its own properties are initialized before it hands off up the chain.

Safety check 2

> A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property. If it doesn't, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.

Safety check 3

> A convenience initializer must delegate to another initializer before assigning a value to *any* property (including properties defined by the same class). If it doesn't, the new value the convenience initializer assigns will be overwritten by its own class's designated initializer.

Safety check 4

An initializer can't call any instance methods, read the values of any instance properties, or refer to `self` as a value until after the first phase of initialization is complete.

The class instance isn't fully valid until the first phase ends. Properties can only be accessed, and methods can only be called, once the class instance is known to be valid at the end of the first phase.

Here's how two-phase initialization plays out, based on the four safety checks above:

Phase 1

- A designated or convenience initializer is called on a class.
- Memory for a new instance of that class is allocated. The memory isn't yet initialized.
- A designated initializer for that class confirms that all stored properties introduced by that class have a value. The memory for these stored properties is now initialized.
- The designated initializer hands off to a superclass initializer to perform the same task for its own stored properties.
- This continues up the class inheritance chain until the top of the chain is reached.
- Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance's memory is considered to be fully initialized, and phase 1 is complete.

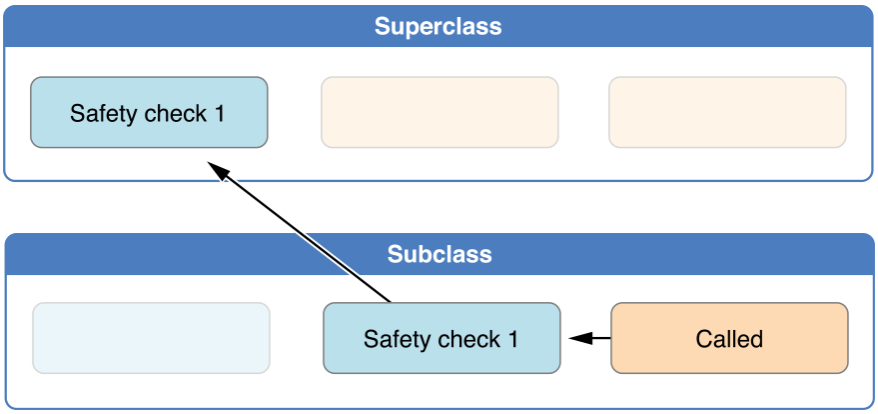
Phase 2

- Working back down from the top of the chain, each designated initializer in the chain has the option to customize the instance further.

Initializers are now able to access `self` and can modify its properties, call its instance methods, and so on.

- Finally, any convenience initializers in the chain have the option to customize the instance and to work with `self`.

Here's how phase 1 looks for an initialization call for a hypothetical subclass and superclass:



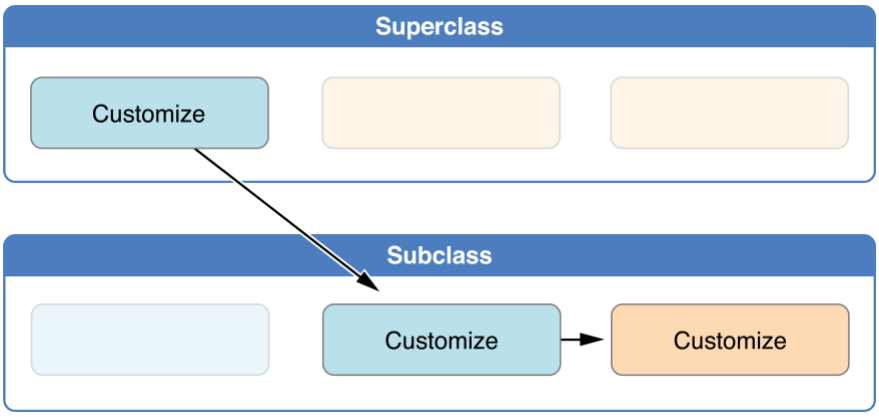
In this example, initialization begins with a call to a convenience initializer on the subclass. This convenience initializer can't yet modify any properties. It delegates across to a designated initializer from the same class.

The designated initializer makes sure that all of the subclass's properties have a value, as per safety check 1. It then calls a designated initializer on its superclass to continue the initialization up the chain.

The superclass's designated initializer makes sure that all of the superclass properties have a value. There are no further superclasses to initialize, and so no further delegation is needed.

As soon as all properties of the superclass have an initial value, its memory is considered fully initialized, and phase 1 is complete.

Here's how phase 2 looks for the same initialization call:



The superclass's designated initializer now has an opportunity to customize the instance further (although it doesn't have to).

Once the superclass's designated initializer is finished, the subclass's designated initializer can perform additional customization (although again, it doesn't have to).

Finally, once the subclass's designated initializer is finished, the convenience initializer that was originally called can perform additional customization.

Initializer Inheritance and Overriding

Unlike subclasses in Objective-C, Swift subclasses don't inherit their superclass initializers by default. Swift's approach prevents a situation in which a simple initializer from a superclass is inherited by a more specialized subclass and is used to create a new instance of the subclass that isn't fully or correctly initialized.

> NOTE
>
> Superclass initializers *are* inherited in certain circumstances, but only when it's safe and appropriate to do so. For more information, see Automatic Initializer Inheritance below.

If you want a custom subclass to present one or more of the same initializers as its superclass, you can provide a custom implementation of those initializers within the subclass.

When you write a subclass initializer that matches a superclass *designated* initializer, you are effectively providing an override of that designated initializer. Therefore, you must write the `override` modifier before the subclass's initializer definition. This is true even if you are overriding an automatically provided default initializer, as described in Default Initializers.

As with an overridden property, method or subscript, the presence of the `override` modifier prompts Swift to check that the superclass has a matching designated initializer to be overridden, and validates that the parameters for your overriding initializer have been specified as intended.

> NOTE
>
> You always write the `override` modifier when overriding a superclass designated initializer, even if your subclass's implementation of the initializer is a convenience initializer.

Conversely, if you write a subclass initializer that matches a superclass *convenience* initializer, that superclass convenience initializer can never be called directly by your subclass, as per the rules described above in Initializer Delegation for Class Types. Therefore, your subclass is not (strictly speaking) providing an override of the superclass initializer. As a result, you don't write the `override` modifier when providing a matching implementation of a superclass convenience initializer.

The example below defines a base class called `Vehicle`. This base class declares a stored property called `numberOfWheels`, with a default `Int` value of `0`. The `numberOfWheels` property is used by a computed property called `description` to create a `String` description of the vehicle's characteristics:

```
class Vehicle {
    var numberOfWheels = 0
    var description: String {
        return "\(numberOfWheels) wheel(s)"
    }
}
```

The `Vehicle` class provides a default value for its only stored property, and doesn't provide any custom initializers itself. As a result, it automatically receives a default initializer, as described in Default Initializers. The default initializer (when available) is always a designated initializer for a class, and can be used to create a new `Vehicle` instance with a `numberOfWheels` of `0`:

```
let vehicle = Vehicle()
print("Vehicle: \(vehicle.description)")
// Vehicle: 0 wheel(s)
```

The next example defines a subclass of `Vehicle` called `Bicycle`:

```
class Bicycle: Vehicle {
    override init() {
        super.init()
        numberOfWheels = 2
    }
}
```

The `Bicycle` subclass defines a custom designated initializer, `init()`. This designated initializer matches a designated initializer from the superclass of

`Bicycle`, and so the `Bicycle` version of this initializer is marked with the `override` modifier.

The `init()` initializer for `Bicycle` starts by calling `super.init()`, which calls the default initializer for the `Bicycle` class's superclass, `Vehicle`. This ensures that the `numberOfWheels` inherited property is initialized by `Vehicle` before `Bicycle` has the opportunity to modify the property. After calling `super.init()`, the original value of `numberOfWheels` is replaced with a new value of `2`.

If you create an instance of `Bicycle`, you can call its inherited `description` computed property to see how its `numberOfWheels` property has been updated:

```
let bicycle = Bicycle()
print("Bicycle: \(bicycle.description)")
// Bicycle: 2 wheel(s)
```

If a subclass initializer performs no customization in phase 2 of the initialization process, and the superclass has a zero-argument designated initializer, you can omit a call to `super.init()` after assigning values to all of the subclass's stored properties.

This example defines another subclass of `Vehicle`, called `Hoverboard`. In its initializer, the `Hoverboard` class sets only its `color` property. Instead of making an explicit call to `super.init()`, this initializer relies on an implicit call to its superclass's initializer to complete the process.

```
class Hoverboard: Vehicle {
    var color: String
    init(color: String) {
        self.color = color
        // super.init() implicitly called here
    }
    override var description: String {
        return "\(super.description) in a beautiful \(color)"
    }
}
```

An instance of `Hoverboard` uses the default number of wheels supplied by the `Vehicle` initializer.

```
let hoverboard = Hoverboard(color: "silver")
print("Hoverboard: \(hoverboard.description)")
// Hoverboard: 0 wheel(s) in a beautiful silver
```

> NOTE
>
> Subclasses can modify inherited variable properties during initialization, but can't modify inherited constant properties.

Automatic Initializer Inheritance

As mentioned above, subclasses don't inherit their superclass initializers by default. However, superclass initializers *are* automatically inherited if certain conditions are met. In practice, this means that you don't need to

write initializer overrides in many common scenarios, and can inherit your superclass initializers with minimal effort whenever it's safe to do so.

Assuming that you provide default values for any new properties you introduce in a subclass, the following two rules apply:

Rule 1

> If your subclass doesn't define any designated initializers, it automatically inherits all of its superclass designated initializers.

Rule 2

> If your subclass provides an implementation of *all* of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

These rules apply even if your subclass adds further convenience initializers.

> NOTE
>
> A subclass can implement a superclass designated initializer as a subclass convenience initializer as part of satisfying rule 2.

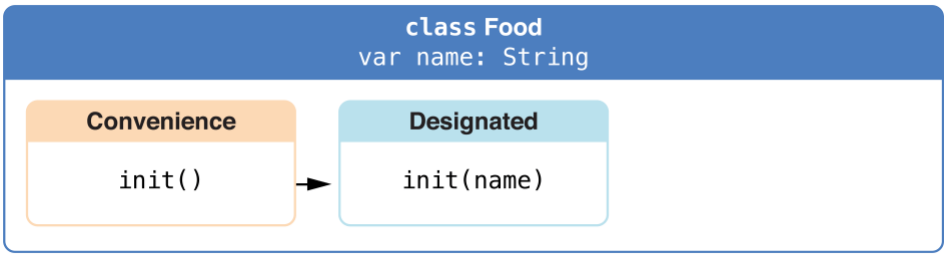
Designated and Convenience Initializers in Action

The following example shows designated initializers, convenience initializers, and automatic initializer inheritance in action. This example defines a hierarchy of three classes called `Food`, `RecipeIngredient`, and `ShoppingListItem`, and demonstrates how their initializers interact.

The base class in the hierarchy is called `Food`, which is a simple class to encapsulate the name of a foodstuff. The `Food` class introduces a single `String` property called `name` and provides two initializers for creating `Food` instances:

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

The figure below shows the initializer chain for the `Food` class:



Classes don't have a default memberwise initializer, and so the `Food` class provides a designated initializer that takes a single argument called `name`. This initializer can be used to create a new `Food` instance with a specific name:

```
let namedMeat = Food(name: "Bacon")
// namedMeat's name is "Bacon"
```

The `init(name: String)` initializer from the `Food` class is provided as a *designated* initializer, because it ensures that all stored properties of a new `Food` instance are fully initialized. The `Food` class doesn't have a superclass,

and so the `init(name: String)` initializer doesn't need to call `super.init()` to complete its initialization.

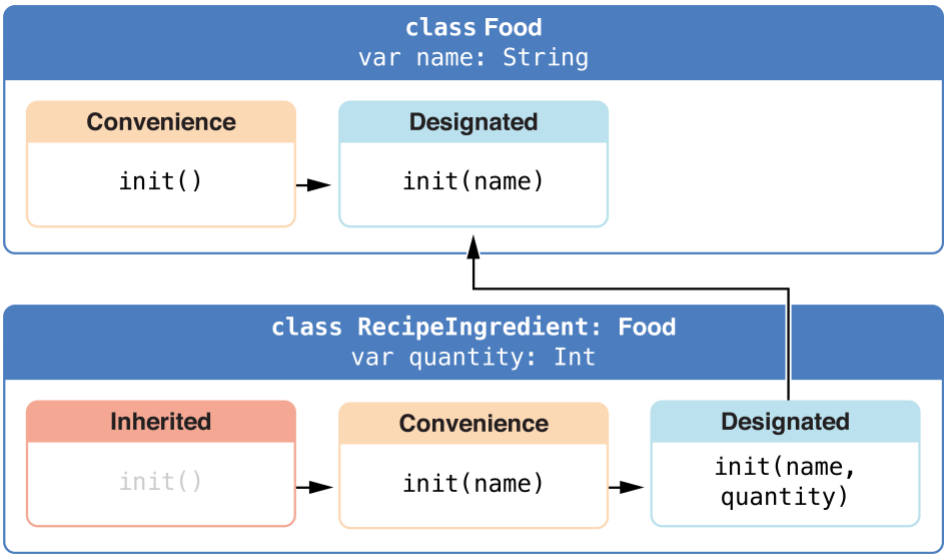
The `Food` class also provides a *convenience* initializer, `init()`, with no arguments. The `init()` initializer provides a default placeholder name for a new food by delegating across to the `Food` class's `init(name: String)` with a `name` value of `[Unnamed]`:

```
let mysteryMeat = Food()
// mysteryMeat's name is "[Unnamed]"
```

The second class in the hierarchy is a subclass of `Food` called `RecipeIngredient`. The `RecipeIngredient` class models an ingredient in a cooking recipe. It introduces an `Int` property called `quantity` (in addition to the `name` property it inherits from `Food`) and defines two initializers for creating `RecipeIngredient` instances:

```
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

The figure below shows the initializer chain for the `RecipeIngredient` class:



The `RecipeIngredient` class has a single designated initializer, `init(name: String, quantity: Int)`, which can be used to populate all of the properties of a new `RecipeIngredient` instance. This initializer starts by assigning the passed `quantity` argument to the `quantity` property, which is the only new property introduced by `RecipeIngredient`. After doing so, the initializer delegates up to the `init(name: String)` initializer of the `Food` class. This process satisfies safety check 1 from [Two-Phase Initialization]() above.

`RecipeIngredient` also defines a convenience initializer, `init(name: String)`, which is used to create a `RecipeIngredient` instance by name alone. This convenience initializer assumes a quantity of `1` for any `RecipeIngredient` instance that's created without an explicit quantity. The definition of this convenience initializer makes `RecipeIngredient` instances quicker and more convenient to create, and avoids code duplication when creating several single-quantity `RecipeIngredient` instances. This convenience initializer simply delegates across to the class's designated initializer, passing in a `quantity` value of `1`.

The `init(name: String)` convenience initializer provided by `RecipeIngredient` takes the same parameters as the `init(name: String)` *designated* initializer from `Food`. Because this convenience initializer overrides a designated initializer from its superclass, it must be

marked with the `override` modifier (as described in [Initializer Inheritance and Overriding](#)).

Even though `RecipeIngredient` provides the `init(name: String)` initializer as a convenience initializer, `RecipeIngredient` has nonetheless provided an implementation of all of its superclass's designated initializers. Therefore, `RecipeIngredient` automatically inherits all of its superclass's convenience initializers too.

In this example, the superclass for `RecipeIngredient` is `Food`, which has a single convenience initializer called `init()`. This initializer is therefore inherited by `RecipeIngredient`. The inherited version of `init()` functions in exactly the same way as the `Food` version, except that it delegates to the `RecipeIngredient` version of `init(name: String)` rather than the `Food` version.

All three of these initializers can be used to create new `RecipeIngredient` instances:

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs",
     quantity: 6)
```

The third and final class in the hierarchy is a subclass of `RecipeIngredient` called `ShoppingListItem`. The `ShoppingListItem` class models a recipe ingredient as it appears in a shopping list.

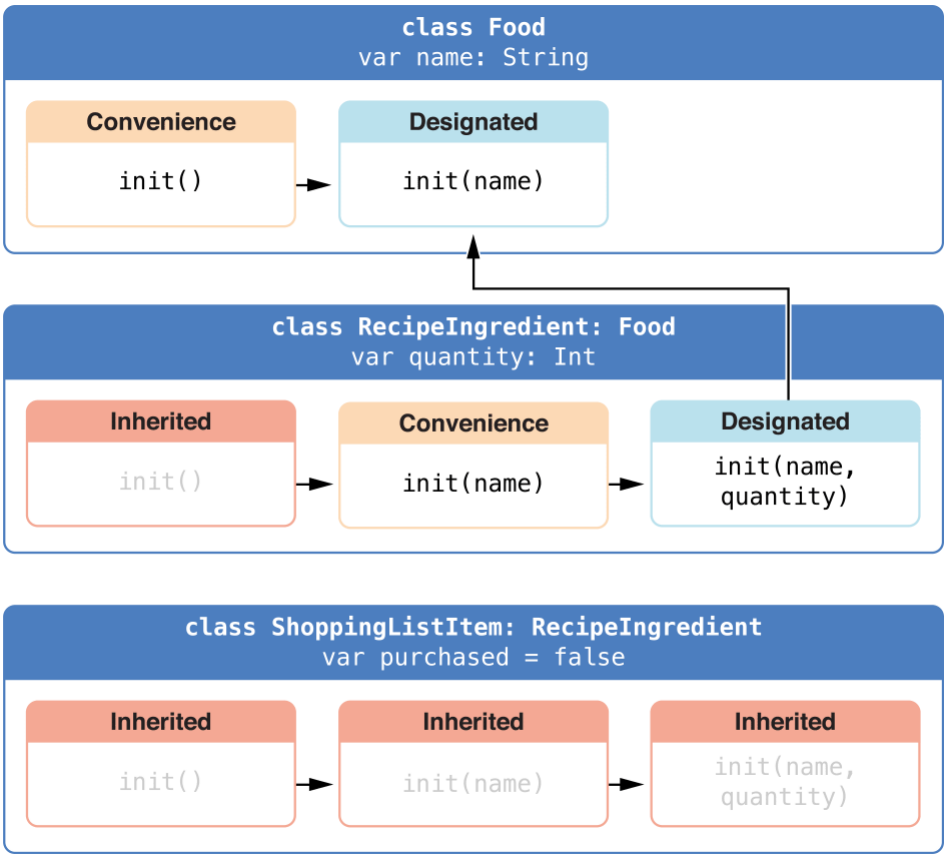
Every item in the shopping list starts out as "unpurchased". To represent this fact, `ShoppingListItem` introduces a Boolean property called `purchased`, with a default value of `false`. `ShoppingListItem` also adds a computed `description` property, which provides a textual description of a `ShoppingListItem` instance:

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

> NOTE
>
> `ShoppingListItem` doesn't define an initializer to provide an initial value for `purchased`, because items in a shopping list (as modeled here) always start out unpurchased.

Because it provides a default value for all of the properties it introduces and doesn't define any initializers itself, `ShoppingListItem` automatically inherits *all* of the designated and convenience initializers from its superclass.

The figure below shows the overall initializer chain for all three classes:



You can use all three of the inherited initializers to create a new `ShoppingListItem` instance:

```
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    print(item.description)
}
// 1 x Orange juice ✔
// 1 x Bacon ✘
// 6 x Eggs ✘
```

Here, a new array called `breakfastList` is created from an array literal containing three new `ShoppingListItem` instances. The type of the array is inferred to be `[ShoppingListItem]`. After the array is created, the name of the `ShoppingListItem` at the start of the array is changed from `"[Unnamed]"` to `"Orange juice"` and it's marked as having been purchased. Printing the description of each item in the array shows that their default states have been set as expected.

Failable Initializers

It's sometimes useful to define a class, structure, or enumeration for which initialization can fail. This failure might be triggered by invalid initialization parameter values, the absence of a required external resource, or some other condition that prevents initialization from succeeding.

To cope with initialization conditions that can fail, define one or more failable initializers as part of a class, structure, or enumeration definition. You write a failable initializer by placing a question mark after the `init` keyword (`init?`).

> NOTE
>
> You can't define a failable and a nonfailable initializer with the same parameter types and names.

A failable initializer creates an *optional* value of the type it initializes. You write `return nil` within a failable initializer to indicate a point at which initialization failure can be triggered.

> NOTE
>
> Strictly speaking, initializers don't return a value. Rather, their role is to ensure that `self` is fully and correctly initialized by the time that initialization ends. Although you write `return nil` to trigger an initialization failure, you don't use the `return` keyword to indicate initialization success.

For instance, failable initializers are implemented for numeric type conversions. To ensure conversion between numeric types maintains the value exactly, use the `init(exactly:)` initializer. If the type conversion can't maintain the value, the initializer fails.

```
let wholeNumber: Double = 12345.0
let pi = 3.14159

if let valueMaintained = Int(exactly: wholeNumber) {
    print("\(wholeNumber) conversion to Int maintains value of \(valueMaintained)")
}
// Prints "12345.0 conversion to Int maintains value of 12345"

let valueChanged = Int(exactly: pi)
// valueChanged is of type Int?, not Int

if valueChanged == nil {
    print("\(pi) conversion to Int doesn't maintain value")
}
// Prints "3.14159 conversion to Int doesn't maintain value"
```

The example below defines a structure called `Animal`, with a constant `String` property called `species`. The `Animal` structure also defines a failable initializer with a single parameter called `species`. This initializer checks if the `species` value passed to the initializer is an empty string. If an empty string is found, an initialization failure is triggered. Otherwise, the `species` property's value is set, and initialization succeeds:

```swift
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}
```

You can use this failable initializer to try to initialize a new `Animal` instance and to check if initialization succeeded:

```swift
let someCreature = Animal(species: "Giraffe")
// someCreature is of type Animal?, not Animal

if let giraffe = someCreature {
    print("An animal was initialized with a species of \(giraffe.species)")
}
// Prints "An animal was initialized with a species of Giraffe"
```

If you pass an empty string value to the failable initializer's `species` parameter, the initializer triggers an initialization failure:

```
let anonymousCreature = Animal(species: "")
// anonymousCreature is of type Animal?, not Animal

if anonymousCreature == nil {
    print("The anonymous creature couldn't be initialized")
}
// Prints "The anonymous creature couldn't be initialized"
```

> NOTE
>
> Checking for an empty string value (such as `""` rather than `"Giraffe"`) isn't the same as checking for `nil` to indicate the absence of an *optional* `String` value. In the example above, an empty string (`""`) is a valid, non-optional `String`. However, it's not appropriate for an animal to have an empty string as the value of its `species` property. To model this restriction, the failable initializer triggers an initialization failure if an empty string is found.

Failable Initializers for Enumerations

You can use a failable initializer to select an appropriate enumeration case based on one or more parameters. The initializer can then fail if the provided parameters don't match an appropriate enumeration case.

The example below defines an enumeration called `TemperatureUnit`, with three possible states (`kelvin`, `celsius`, and `fahrenheit`). A failable initializer is used to find an appropriate enumeration case for a `Character` value representing a temperature symbol:

```swift
enum TemperatureUnit {
    case kelvin, celsius, fahrenheit
    init?(symbol: Character) {
        switch symbol {
        case "K":
            self = .kelvin
        case "C":
            self = .celsius
        case "F":
            self = .fahrenheit
        default:
            return nil
        }
    }
}
```

You can use this failable initializer to choose an appropriate enumeration case for the three possible states and to cause initialization to fail if the parameter doesn't match one of these states:

```
let fahrenheitUnit = TemperatureUnit(symbol: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// Prints "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(symbol: "X")
if unknownUnit == nil {
    print("This isn't a defined temperature unit, so initialization failed.")
}
// Prints "This isn't a defined temperature unit, so initialization failed."
```

Failable Initializers for Enumerations with Raw Values

Enumerations with raw values automatically receive a failable initializer, `init?(rawValue:)`, that takes a parameter called `rawValue` of the appropriate raw-value type and selects a matching enumeration case if one is found, or triggers an initialization failure if no matching value exists.

You can rewrite the `TemperatureUnit` example from above to use raw values of type `Character` and to take advantage of the `init?(rawValue:)` initializer:

```
enum TemperatureUnit: Character {
    case kelvin = "K", celsius = "C", fahrenheit = "F"
}

let fahrenheitUnit = TemperatureUnit(rawValue: "F")
if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so initialization succeeded.")
}
// Prints "This is a defined temperature unit, so initialization succeeded."

let unknownUnit = TemperatureUnit(rawValue: "X")
if unknownUnit == nil {
    print("This isn't a defined temperature unit, so initialization failed.")
}
// Prints "This isn't a defined temperature unit, so initialization failed."
```

Propagation of Initialization Failure

A failable initializer of a class, structure, or enumeration can delegate across to another failable initializer from the same class, structure, or enumeration. Similarly, a subclass failable initializer can delegate up to a superclass failable initializer.

In either case, if you delegate to another initializer that causes initialization to fail, the entire initialization process fails immediately, and no further initialization code is executed.

> NOTE
>
> A failable initializer can also delegate to a nonfailable initializer. Use this approach if you need to add a potential failure state to an existing initialization process that doesn't otherwise fail.

The example below defines a subclass of `Product` called `CartItem`. The `CartItem` class models an item in an online shopping cart. `CartItem` introduces a stored constant property called `quantity` and ensures that this property always has a value of at least `1`:

```
class Product {
    let name: String
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}

class CartItem: Product {
    let quantity: Int
    init?(name: String, quantity: Int) {
        if quantity < 1 { return nil }
        self.quantity = quantity
        super.init(name: name)
    }
}
```

The failable initializer for `CartItem` starts by validating that it has received a `quantity` value of `1` or more. If the `quantity` is invalid, the entire initialization process fails immediately and no further initialization code is executed. Likewise, the failable initializer for `Product` checks the `name` value, and the initializer process fails immediately if `name` is the empty string.

If you create a `CartItem` instance with a nonempty name and a quantity of `1` or more, initialization succeeds:

```
if let twoSocks = CartItem(name: "sock", quantity: 2) {
    print("Item: \(twoSocks.name), quantity: \(twoSocks.quantity)")
}
// Prints "Item: sock, quantity: 2"
```

If you try to create a `CartItem` instance with a `quantity` value of `0`, the `CartItem` initializer causes initialization to fail:

```
if let zeroShirts = CartItem(name: "shirt", quantity: 0) {
    print("Item: \(zeroShirts.name), quantity: \(zeroShirts.quantity)")
} else {
    print("Unable to initialize zero shirts")
}
// Prints "Unable to initialize zero shirts"
```

Similarly, if you try to create a `CartItem` instance with an empty `name` value, the superclass `Product` initializer causes initialization to fail:

```
if let oneUnnamed = CartItem(name: "", quantity: 1) {
    print("Item: \(oneUnnamed.name), quantity: \(oneUnnamed.quantity)")
} else {
    print("Unable to initialize one unnamed product")
}
// Prints "Unable to initialize one unnamed product"
```

Overriding a Failable Initializer

You can override a superclass failable initializer in a subclass, just like any other initializer. Alternatively, you can override a superclass failable initializer with a subclass *nonfailable* initializer. This enables you to define a subclass for which initialization can't fail, even though initialization of the superclass is allowed to fail.

Note that if you override a failable superclass initializer with a nonfailable subclass initializer, the only way to delegate up to the superclass initializer is to force-unwrap the result of the failable superclass initializer.

> NOTE
>
> You can override a failable initializer with a nonfailable initializer but not the other way around.

The example below defines a class called `Document`. This class models a document that can be initialized with a `name` property that's either a nonempty string value or `nil`, but can't be an empty string:

```
class Document {
    var name: String?
    // this initializer creates a document with a nil name value
    init() {}
    // this initializer creates a document with a nonempty name value
    init?(name: String) {
        if name.isEmpty { return nil }
        self.name = name
    }
}
```

The next example defines a subclass of `Document` called `AutomaticallyNamedDocument`. The `AutomaticallyNamedDocument` subclass overrides both of the designated initializers introduced by `Document`. These overrides ensure that an `AutomaticallyNamedDocument` instance has an initial `name` value of `"[Untitled]"` if the instance is initialized without a name, or if an empty string is passed to the `init(name:)` initializer:

```
class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
    override init(name: String) {
        super.init()
        if name.isEmpty {
            self.name = "[Untitled]"
        } else {
            self.name = name
        }
    }
}
```

The `AutomaticallyNamedDocument` overrides its superclass's failable `init?(name:)` initializer with a nonfailable `init(name:)` initializer. Because `AutomaticallyNamedDocument` copes with the empty string case in a different way than its superclass, its initializer doesn't need to fail, and so it provides a nonfailable version of the initializer instead.

You can use forced unwrapping in an initializer to call a failable initializer from the superclass as part of the implementation of a subclass's nonfailable initializer. For example, the `UntitledDocument` subclass below is always named `"[Untitled]"`, and it uses the failable `init(name:)` initializer from its superclass during initialization.

```
class UntitledDocument: Document {
    override init() {
        super.init(name: "[Untitled]")!
    }
}
```

In this case, if the `init(name:)` initializer of the superclass were ever called with an empty string as the name, the forced unwrapping operation would result in a runtime error. However, because it's called with a string constant, you can see that the initializer won't fail, so no runtime error can occur in this case.

The init! Failable Initializer

You typically define a failable initializer that creates an optional instance of the appropriate type by placing a question mark after the `init` keyword (`init?`). Alternatively, you can define a failable initializer that creates an implicitly unwrapped optional instance of the appropriate type. Do this by placing an exclamation point after the `init` keyword (`init!`) instead of a question mark.

You can delegate from `init?` to `init!` and vice versa, and you can override `init?` with `init!` and vice versa. You can also delegate from `init` to `init!`, although doing so will trigger an assertion if the `init!` initializer causes initialization to fail.

Required Initializers

Write the `required` modifier before the definition of a class initializer to indicate that every subclass of the class must implement that initializer:

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}
```

You must also write the `required` modifier before every subclass implementation of a required initializer, to indicate that the initializer requirement applies to further subclasses in the chain. You don't write the `override` modifier when overriding a required designated initializer:

```
class SomeSubclass: SomeClass {
    required init() {
        // subclass implementation of the required initializer goes here
    }
}
```

> NOTE
>
> You don't have to provide an explicit implementation of a required initializer if you can satisfy the requirement with an inherited initializer.

Setting a Default Property Value with a Closure or Function

If a stored property's default value requires some customization or setup, you can use a closure or global function to provide a customized default

value for that property. Whenever a new instance of the type that the property belongs to is initialized, the closure or function is called, and its return value is assigned as the property's default value.

These kinds of closures or functions typically create a temporary value of the same type as the property, tailor that value to represent the desired initial state, and then return that temporary value to be used as the property's default value.

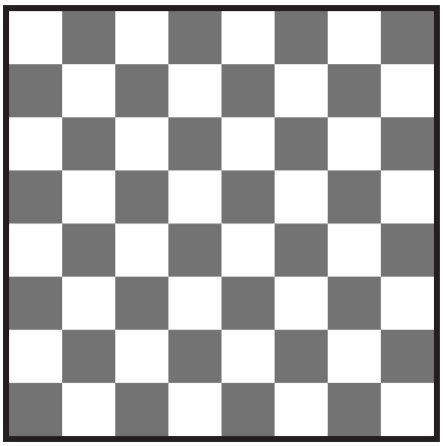
Here's a skeleton outline of how a closure can be used to provide a default property value:

```
class SomeClass {
    let someProperty: SomeType = {
        // create a default value for someProperty inside this closure
        // someValue must be of the same type as SomeType
        return someValue
    }()
}
```

Note that the closure's end curly brace is followed by an empty pair of parentheses. This tells Swift to execute the closure immediately. If you omit these parentheses, you are trying to assign the closure itself to the property, and not the return value of the closure.

> NOTE
>
> If you use a closure to initialize a property, remember that the rest of the instance hasn't yet been initialized at the point that the closure is executed. This means that you can't access any other property values from within your closure, even if those properties have default values. You also can't use the implicit `self` property, or call any of the instance's methods.

The example below defines a structure called `Chessboard`, which models a board for the game of chess. Chess is played on an 8 x 8 board, with alternating black and white squares.



To represent this game board, the `Chessboard` structure has a single property called `boardColors`, which is an array of 64 `Bool` values. A value of `true` in the array represents a black square and a value of `false` represents a white square. The first item in the array represents the top left square on the board and the last item in the array represents the bottom right square on the board.

The `boardColors` array is initialized with a closure to set up its color values:

```
struct Chessboard {
    let boardColors: [Bool] = {
        var temporaryBoard: [Bool] = []
        var isBlack = false
        for i in 1...8 {
            for j in 1...8 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
    }()
    func squareIsBlackAt(row: Int, column: Int) -> Bool {
        return boardColors[(row * 8) + column]
    }
}
```

Whenever a new `Chessboard` instance is created, the closure is executed, and the default value of `boardColors` is calculated and returned. The closure in the example above calculates and sets the appropriate color for each square on the board in a temporary array called `temporaryBoard`, and returns this temporary array as the closure's return value once its setup is complete. The returned array value is stored in `boardColors` and can be queried with the `squareIsBlackAt(row:column:)` utility function:

```swift
let board = Chessboard()
print(board.squareIsBlackAt(row: 0, column: 1))
// Prints "true"
print(board.squareIsBlackAt(row: 7, column: 7))
// Prints "false"
```

Deinitialization

A *deinitializer* is called immediately before a class instance is deallocated. You write deinitializers with the `deinit` keyword, similar to how initializers are written with the `init` keyword. Deinitializers are only available on class types.

How Deinitialization Works

Swift automatically deallocates your instances when they're no longer needed, to free up resources. Swift handles the memory management of instances through *automatic reference counting* (*ARC*), as described in Automatic Reference Counting. Typically you don't need to perform manual cleanup when your instances are deallocated. However, when you are working with your own resources, you might need to perform some additional cleanup yourself. For example, if you create a custom class to open a file and write some data to it, you might need to close the file before the class instance is deallocated.

Class definitions can have at most one deinitializer per class. The deinitializer doesn't take any parameters and is written without parentheses:

```
deinit {
    // perform the deinitialization
}
```

Deinitializers are called automatically, just before instance deallocation takes place. You aren't allowed to call a deinitializer yourself. Superclass deinitializers are inherited by their subclasses, and the superclass deinitializer is called automatically at the end of a subclass deinitializer

implementation. Superclass deinitializers are always called, even if a subclass doesn't provide its own deinitializer.

Because an instance isn't deallocated until after its deinitializer is called, a deinitializer can access all properties of the instance it's called on and can modify its behavior based on those properties (such as looking up the name of a file that needs to be closed).

Deinitializers in Action

Here's an example of a deinitializer in action. This example defines two new types, `Bank` and `Player`, for a simple game. The `Bank` class manages a made-up currency, which can never have more than 10,000 coins in circulation. There can only ever be one `Bank` in the game, and so the `Bank` is implemented as a class with type properties and methods to store and manage its current state:

```
class Bank {
    static var coinsInBank = 10_000
    static func distribute(coins
     numberOfCoinsRequested: Int) -> Int {
        let numberOfCoinsToVend =
     min(numberOfCoinsRequested, coinsInBank)
        coinsInBank -= numberOfCoinsToVend
        return numberOfCoinsToVend
    }
    static func receive(coins: Int) {
        coinsInBank += coins
    }
}
```

`Bank` keeps track of the current number of coins it holds with its `coinsInBank` property. It also offers two methods—`distribute(coins:)` and `receive(coins:)`—to handle the distribution and collection of coins.

The `distribute(coins:)` method checks that there are enough coins in the bank before distributing them. If there aren't enough coins, `Bank` returns a smaller number than the number that was requested (and returns zero if no coins are left in the bank). It returns an integer value to indicate the actual number of coins that were provided.

The `receive(coins:)` method simply adds the received number of coins back into the bank's coin store.

The `Player` class describes a player in the game. Each player has a certain number of coins stored in their purse at any time. This is represented by the player's `coinsInPurse` property:

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins:
     coins)
    }
    deinit {
        Bank.receive(coins: coinsInPurse)
    }
}
```

Each `Player` instance is initialized with a starting allowance of a specified number of coins from the bank during initialization, although a `Player` instance may receive fewer than that number if not enough coins are available.

The `Player` class defines a `win(coins:)` method, which retrieves a certain number of coins from the bank and adds them to the player's purse. The `Player` class also implements a deinitializer, which is called just before a `Player` instance is deallocated. Here, the deinitializer simply returns all of the player's coins to the bank:

```
var playerOne: Player? = Player(coins: 100)
print("A new player has joined the game with \
      (playerOne!.coinsInPurse) coins")
// Prints "A new player has joined the game with 100
      coins"
print("There are now \(Bank.coinsInBank) coins left
      in the bank")
// Prints "There are now 9900 coins left in the
      bank"
```

A new `Player` instance is created, with a request for 100 coins if they're available. This `Player` instance is stored in an optional `Player` variable called `playerOne`. An optional variable is used here, because players can leave the game at any point. The optional lets you track whether there's currently a player in the game.

Because `playerOne` is an optional, it's qualified with an exclamation point (`!`) when its `coinsInPurse` property is accessed to print its default number of coins, and whenever its `win(coins:)` method is called:

```
playerOne!.win(coins: 2_000)
print("PlayerOne won 2000 coins & now has \
      (playerOne!.coinsInPurse) coins")
// Prints "PlayerOne won 2000 coins & now has 2100
      coins"
print("The bank now only has \(Bank.coinsInBank)
      coins left")
// Prints "The bank now only has 7900 coins left"
```

Here, the player has won 2,000 coins. The player's purse now contains 2,100 coins, and the bank has only 7,900 coins left.

```
playerOne = nil
print("PlayerOne has left the game")
// Prints "PlayerOne has left the game"
print("The bank now has \(Bank.coinsInBank) coins")
// Prints "The bank now has 10000 coins"
```

The player has now left the game. This is indicated by setting the optional `playerOne` variable to `nil`, meaning "no `Player` instance." At the point that this happens, the `playerOne` variable's reference to the `Player` instance is broken. No other properties or variables are still referring to the `Player` instance, and so it's deallocated in order to free up its memory. Just before this happens, its deinitializer is called automatically, and its coins are returned to the bank.

Optional Chaining

Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be `nil`. If the optional contains a value, the property, method, or subscript call succeeds; if the optional is `nil`, the property, method, or subscript call returns `nil`. Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is `nil`.

> NOTE
>
> Optional chaining in Swift is similar to messaging `nil` in Objective-C, but in a way that works for any type, and that can be checked for success or failure.

Optional Chaining as an Alternative to Forced Unwrapping

You specify optional chaining by placing a question mark (`?`) after the optional value on which you wish to call a property, method or subscript if the optional is non-`nil`. This is very similar to placing an exclamation point (`!`) after an optional value to force the unwrapping of its value. The main difference is that optional chaining fails gracefully when the optional is `nil`, whereas forced unwrapping triggers a runtime error when the optional is `nil`.

To reflect the fact that optional chaining can be called on a `nil` value, the result of an optional chaining call is always an optional value, even if the property, method, or subscript you are querying returns a non-optional value. You can use this optional return value to check whether the optional chaining call was successful (the returned optional contains a value), or didn't succeed due to a `nil` value in the chain (the returned optional value is `nil`).

Specifically, the result of an optional chaining call is of the same type as the expected return value, but wrapped in an optional. A property that normally returns an `Int` will return an `Int?` when accessed through optional chaining.

The next several code snippets demonstrate how optional chaining differs from forced unwrapping and enables you to check for success.

First, two classes called `Person` and `Residence` are defined:

```
class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}
```

`Residence` instances have a single `Int` property called `numberOfRooms`, with a default value of `1`. `Person` instances have an optional `residence` property of type `Residence?`.

If you create a new `Person` instance, its `residence` property is default initialized to `nil`, by virtue of being optional. In the code below, `john` has a `residence` property value of `nil`:

```
let john = Person()
```

If you try to access the `numberOfRooms` property of this person's `residence`, by placing an exclamation point after `residence` to force the unwrapping of its value, you trigger a runtime error, because there's no `residence` value to unwrap:

```
let roomCount = john.residence!.numberOfRooms
// this triggers a runtime error
```

The code above succeeds when `john.residence` has a non-`nil` value and will set `roomCount` to an `Int` value containing the appropriate number of rooms. However, this code always triggers a runtime error when `residence` is `nil`, as illustrated above.

Optional chaining provides an alternative way to access the value of `numberOfRooms`. To use optional chaining, use a question mark in place of the exclamation point:

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

This tells Swift to “chain” on the optional `residence` property and to retrieve the value of `numberOfRooms` if `residence` exists.

Because the attempt to access `numberOfRooms` has the potential to fail, the optional chaining attempt returns a value of type `Int?`, or “optional `Int`”. When `residence` is `nil`, as in the example above, this optional `Int` will also be `nil`, to reflect the fact that it was not possible to access `numberOfRooms`. The optional `Int` is accessed through optional binding to unwrap the integer and assign the non-optional value to the `roomCount` constant.

Note that this is true even though `numberOfRooms` is a non-optional `Int`. The fact that it's queried through an optional chain means that the call to `numberOfRooms` will always return an `Int?` instead of an `Int`.

You can assign a `Residence` instance to `john.residence`, so that it no longer has a `nil` value:

```
john.residence = Residence()
```

`john.residence` now contains an actual `Residence` instance, rather than `nil`. If you try to access `numberOfRooms` with the same optional chaining as before, it will now return an `Int?` that contains the default `numberOfRooms` value of `1`:

```
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "John's residence has 1 room(s)."
```

Defining Model Classes for Optional Chaining

You can use optional chaining with calls to properties, methods, and subscripts that are more than one level deep. This enables you to drill down into subproperties within complex models of interrelated types, and to check whether it's possible to access properties, methods, and subscripts on those subproperties.

The code snippets below define four model classes for use in several subsequent examples, including examples of multilevel optional chaining. These classes expand upon the `Person` and `Residence` model from above by adding a `Room` and `Address` class, with associated properties, methods, and subscripts.

The `Person` class is defined in the same way as before:

```
class Person {
    var residence: Residence?
}
```

The `Residence` class is more complex than before. This time, the `Residence` class defines a variable property called `rooms`, which is initialized with an empty array of type `[Room]`:

```
class Residence {
    var rooms: [Room] = []
    var numberOfRooms: Int {
        return rooms.count
    }
    subscript(i: Int) -> Room {
        get {
            return rooms[i]
        }
        set {
            rooms[i] = newValue
        }
    }
    func printNumberOfRooms() {
        print("The number of rooms is \
     (numberOfRooms)")
    }
    var address: Address?
}
```

Because this version of `Residence` stores an array of `Room` instances, its `numberOfRooms` property is implemented as a computed property, not a stored property. The computed `numberOfRooms` property simply returns the value of the `count` property from the `rooms` array.

As a shortcut to accessing its `rooms` array, this version of `Residence` provides a read-write subscript that provides access to the room at the requested index in the `rooms` array.

This version of `Residence` also provides a method called `printNumberOfRooms`, which simply prints the number of rooms in the residence.

Finally, `Residence` defines an optional property called `address`, with a type of `Address?`. The `Address` class type for this property is defined below.

The `Room` class used for the `rooms` array is a simple class with one property called `name`, and an initializer to set that property to a suitable room name:

```
class Room {
    let name: String
    init(name: String) { self.name = name }
}
```

The final class in this model is called `Address`. This class has three optional properties of type `String?`. The first two properties, `buildingName` and `buildingNumber`, are alternative ways to identify a particular building as part of an address. The third property, `street`, is used to name the street for that address:

```
class Address {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if let buildingNumber = buildingNumber, let street = street {
            return "\(buildingNumber) \(street)"
        } else if buildingName != nil {
            return buildingName
        } else {
            return nil
        }
    }
}
```

The `Address` class also provides a method called `buildingIdentifier()`, which has a return type of `String?`. This method checks the properties of the address and returns `buildingName` if it has a value, or `buildingNumber` concatenated with `street` if both have values, or `nil` otherwise.

Accessing Properties Through Optional Chaining

As demonstrated in [Optional Chaining as an Alternative to Forced Unwrapping](), you can use optional chaining to access a property on an optional value, and to check if that property access is successful.

Use the classes defined above to create a new `Person` instance, and try to access its `numberOfRooms` property as before:

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    print("John's residence has \(roomCount) room(s).")
} else {
    print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

Because `john.residence` is `nil`, this optional chaining call fails in the same way as before.

You can also attempt to set a property's value through optional chaining:

```
let someAddress = Address()
someAddress.buildingNumber = "29"
someAddress.street = "Acacia Road"
john.residence?.address = someAddress
```

In this example, the attempt to set the `address` property of `john.residence` will fail, because `john.residence` is currently `nil`.

The assignment is part of the optional chaining, which means none of the code on the right-hand side of the `=` operator is evaluated. In the previous example, it's not easy to see that `someAddress` is never evaluated, because accessing a constant doesn't have any side effects. The listing below does the same assignment, but it uses a function to create the address. The

function prints “Function was called” before returning a value, which lets you see whether the right-hand side of the = operator was evaluated.

```
1  func createAddress() -> Address {
2      print("Function was called.")
3
4      let someAddress = Address()
5      someAddress.buildingNumber = "29"
6      someAddress.street = "Acacia Road"
7
8      return someAddress
9  }
10 john.residence?.address = createAddress()
```

You can tell that the `createAddress()` function isn’t called, because nothing is printed.

Calling Methods Through Optional Chaining

You can use optional chaining to call a method on an optional value, and to check whether that method call is successful. You can do this even if that method doesn’t define a return value.

The `printNumberOfRooms()` method on the `Residence` class prints the current value of `numberOfRooms`. Here’s how the method looks:

```
func printNumberOfRooms() {
    print("The number of rooms is \(numberOfRooms)")
}
```

This method doesn't specify a return type. However, functions and methods with no return type have an implicit return type of `Void`, as described in [Functions Without Return Values]. This means that they return a value of `()`, or an empty tuple.

If you call this method on an optional value with optional chaining, the method's return type will be `Void?`, not `Void`, because return values are always of an optional type when called through optional chaining. This enables you to use an `if` statement to check whether it was possible to call the `printNumberOfRooms()` method, even though the method doesn't itself define a return value. Compare the return value from the `printNumberOfRooms` call against `nil` to see if the method call was successful:

```
if john.residence?.printNumberOfRooms() != nil {
    print("It was possible to print the number of rooms.")
} else {
    print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."
```

The same is true if you attempt to set a property through optional chaining. The example above in [Accessing Properties Through Optional Chaining] attempts to set an `address` value for `john.residence`, even though the

`residence` property is `nil`. Any attempt to set a property through optional chaining returns a value of type `Void?`, which enables you to compare against `nil` to see if the property was set successfully:

```
if (john.residence?.address = someAddress) != nil {
    print("It was possible to set the address.")
} else {
    print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

Accessing Subscripts Through Optional Chaining

You can use optional chaining to try to retrieve and set a value from a subscript on an optional value, and to check whether that subscript call is successful.

> NOTE
>
> When you access a subscript on an optional value through optional chaining, you place the question mark *before* the subscript's brackets, not after. The optional chaining question mark always follows immediately after the part of the expression that's optional.

The example below tries to retrieve the name of the first room in the `rooms` array of the `john.residence` property using the subscript defined on the `Residence` class. Because `john.residence` is currently `nil`, the subscript call fails:

```
if let firstRoomName = john.residence?[0].name {
    print("The first room name is \
     (firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "Unable to retrieve the first room name."
```

The optional chaining question mark in this subscript call is placed immediately after `john.residence`, before the subscript brackets, because `john.residence` is the optional value on which optional chaining is being attempted.

Similarly, you can try to set a new value through a subscript with optional chaining:

```
john.residence?[0] = Room(name: "Bathroom")
```

This subscript setting attempt also fails, because `residence` is currently `nil`.

If you create and assign an actual `Residence` instance to `john.residence`, with one or more `Room` instances in its `rooms` array, you can use the `Residence` subscript to access the actual items in the `rooms` array through optional chaining:

```
let johnsHouse = Residence()
johnsHouse.rooms.append(Room(name: "Living Room"))
johnsHouse.rooms.append(Room(name: "Kitchen"))
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    print("The first room name is \
     (firstRoomName).")
} else {
    print("Unable to retrieve the first room name.")
}
// Prints "The first room name is Living Room."
```

Accessing Subscripts of Optional Type

If a subscript returns a value of optional type—such as the key subscript of Swift's `Dictionary` type—place a question mark *after* the subscript's closing bracket to chain on its optional return value:

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79,
     94, 81]]
testScores["Dave"]?[0] = 91
testScores["Bev"]?[0] += 1
testScores["Brian"]?[0] = 72
// the "Dave" array is now [91, 82, 84] and the
     "Bev" array is now [80, 94, 81]
```

The example above defines a dictionary called `testScores`, which contains two key-value pairs that map a `String` key to an array of `Int` values. The example uses optional chaining to set the first item in the `"Dave"` array to `91`; to increment the first item in the `"Bev"` array by `1`; and to try to set the first item in an array for a key of `"Brian"`. The first two calls succeed, because the `testScores` dictionary contains keys for `"Dave"` and `"Bev"`. The third call fails, because the `testScores` dictionary doesn't contain a key for `"Brian"`.

Linking Multiple Levels of Chaining

You can link together multiple levels of optional chaining to drill down to properties, methods, and subscripts deeper within a model. However, multiple levels of optional chaining don't add more levels of optionality to the returned value.

To put it another way:

- If the type you are trying to retrieve isn't optional, it will become optional because of the optional chaining.
- If the type you are trying to retrieve is *already* optional, it will not become *more* optional because of the chaining.

Therefore:

- If you try to retrieve an `Int` value through optional chaining, an `Int?` is always returned, no matter how many levels of chaining are used.
- Similarly, if you try to retrieve an `Int?` value through optional chaining, an `Int?` is always returned, no matter how many levels of chaining are used.

The example below tries to access the `street` property of the `address` property of the `residence` property of `john`. There are *two* levels of optional chaining in use here, to chain through the `residence` and `address` properties, both of which are of optional type:

```
if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "Unable to retrieve the address."
```

The value of `john.residence` currently contains a valid `Residence` instance. However, the value of `john.residence.address` is currently `nil`. Because of this, the call to `john.residence?.address?.street` fails.

Note that in the example above, you are trying to retrieve the value of the `street` property. The type of this property is `String?`. The return value of `john.residence?.address?.street` is therefore also `String?`, even though two levels of optional chaining are applied in addition to the underlying optional type of the property.

If you set an actual `Address` instance as the value for `john.residence.address`, and set an actual value for the address's `street` property, you can access the value of the `street` property through multilevel optional chaining:

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence?.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {
    print("John's street name is \(johnsStreet).")
} else {
    print("Unable to retrieve the address.")
}
// Prints "John's street name is Laurel Street."
```

In this example, the attempt to set the `address` property of `john.residence` will succeed, because the value of `john.residence` currently contains a valid `Residence` instance.

Chaining on Methods with Optional Return Values

The previous example shows how to retrieve the value of a property of optional type through optional chaining. You can also use optional chaining to call a method that returns a value of optional type, and to chain on that method's return value if needed.

The example below calls the `Address` class's `buildingIdentifier()` method through optional chaining. This method returns a value of type `String?`. As described above, the ultimate return type of this method call after optional chaining is also `String?`:

```
if let buildingIdentifier = john.residence?.address?.buildingIdentifier() {
    print("John's building identifier is \(buildingIdentifier).")
}
// Prints "John's building identifier is The Larches."
```

If you want to perform further optional chaining on this method's return value, place the optional chaining question mark *after* the method's parentheses:

```
if let beginsWithThe =

    john.residence?.address?.buildingIdentifier()?.hasPrefix("The") {
    if beginsWithThe {
        print("John's building identifier begins with \"The\".")
    } else {
        print("John's building identifier doesn't begin with \"The\".")
    }
}
// Prints "John's building identifier begins with "The"."
```

> NOTE
>
> In the example above, you place the optional chaining question mark *after* the parentheses, because the optional value you are chaining on is the `buildingIdentifier()` method's return value, and not the `buildingIdentifier()` method itself.

Error Handling

Error handling is the process of responding to and recovering from error conditions in your program. Swift provides first-class support for throwing, catching, propagating, and manipulating recoverable errors at runtime.

Some operations aren't guaranteed to always complete execution or produce a useful output. Optionals are used to represent the absence of a value, but when an operation fails, it's often useful to understand what caused the failure, so that your code can respond accordingly.

As an example, consider the task of reading and processing data from a file on disk. There are a number of ways this task can fail, including the file not existing at the specified path, the file not having read permissions, or the file not being encoded in a compatible format. Distinguishing among these different situations allows a program to resolve some errors and to communicate to the user any errors it can't resolve.

> NOTE
>
> Error handling in Swift interoperates with error handling patterns that use the `NSError` class in Cocoa and Objective-C. For more information about this class, see Handling Cocoa Errors in Swift.

Representing and Throwing Errors

In Swift, errors are represented by values of types that conform to the `Error` protocol. This empty protocol indicates that a type can be used for error handling.

Swift enumerations are particularly well suited to modeling a group of related error conditions, with associated values allowing for additional

information about the nature of an error to be communicated. For example, here's how you might represent the error conditions of operating a vending machine inside a game:

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

Throwing an error lets you indicate that something unexpected happened and the normal flow of execution can't continue. You use a `throw` statement to throw an error. For example, the following code throws an error to indicate that five additional coins are needed by the vending machine:

```
throw
     VendingMachineError.insufficientFunds(coinsNeed
     ed: 5)
```

Handling Errors

When an error is thrown, some surrounding piece of code must be responsible for handling the error—for example, by correcting the problem, trying an alternative approach, or informing the user of the failure.

There are four ways to handle errors in Swift. You can propagate the error from a function to the code that calls that function, handle the error using a `do-catch` statement, handle the error as an optional value, or assert that the error will not occur. Each approach is described in a section below.

When a function throws an error, it changes the flow of your program, so it's important that you can quickly identify places in your code that can throw errors. To identify these places in your code, write the `try` keyword —or the `try?` or `try!` variation—before a piece of code that calls a function, method, or initializer that can throw an error. These keywords are described in the sections below.

> NOTE
>
> Error handling in Swift resembles exception handling in other languages, with the use of the `try`, `catch` and `throw` keywords. Unlike exception handling in many languages—including Objective-C—error handling in Swift doesn't involve unwinding the call stack, a process that can be computationally expensive. As such, the performance characteristics of a `throw` statement are comparable to those of a `return` statement.

Propagating Errors Using Throwing Functions

To indicate that a function, method, or initializer can throw an error, you write the `throws` keyword in the function's declaration after its parameters. A function marked with `throws` is called a *throwing function*. If the function specifies a return type, you write the `throws` keyword before the return arrow (`->`).

```
func canThrowErrors() throws -> String

func cannotThrowErrors() -> String
```

A throwing function propagates errors that are thrown inside of it to the scope from which it's called.

> NOTE
>
> Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside the function.

In the example below, the `VendingMachine` class has a `vend(itemNamed:)` method that throws an appropriate `VendingMachineError` if the requested item isn't available, is out of stock, or has a cost that exceeds the current deposited amount:

```
1  struct Item {
2      var price: Int
3      var count: Int
4  }
5
6  class VendingMachine {
7      var inventory = [
8          "Candy Bar": Item(price: 12, count: 7),
9          "Chips": Item(price: 10, count: 4),
10         "Pretzels": Item(price: 7, count: 11)
11     ]
12     var coinsDeposited = 0
13
14     func vend(itemNamed name: String) throws {
15         guard let item = inventory[name] else {
16             throw
       VendingMachineError.invalidSelection
17         }
18
19         guard item.count > 0 else {
20             throw VendingMachineError.outOfStock
21         }
22
23         guard item.price <= coinsDeposited else {
24             throw
       VendingMachineError.insufficientFunds(coinsNeed
```

```
        ed: item.price - coinsDeposited)
        }

        coinsDeposited -= item.price

        var newItem = item
        newItem.count -= 1
        inventory[name] = newItem

        print("Dispensing \(name)")
    }
}
```

The implementation of the `vend(itemNamed:)` method uses `guard` statements to exit the method early and throw appropriate errors if any of the requirements for purchasing a snack aren't met. Because a `throw` statement immediately transfers program control, an item will be vended only if all of these requirements are met.

Because the `vend(itemNamed:)` method propagates any errors it throws, any code that calls this method must either handle the errors—using a `do-catch` statement, `try?`, or `try!`—or continue to propagate them. For example, the `buyFavoriteSnack(person:vendingMachine:)` in the example below is also a throwing function, and any errors that the `vend(itemNamed:)` method throws will propagate up to the point where the `buyFavoriteSnack(person:vendingMachine:)` function is called.

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]
func buyFavoriteSnack(person: String,
     vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy
     Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
```

In this example, the `buyFavoriteSnack(person: vendingMachine:)` function looks up a given person's favorite snack and tries to buy it for them by calling the `vend(itemNamed:)` method. Because the `vend(itemNamed:)` method can throw an error, it's called with the `try` keyword in front of it.

Throwing initializers can propagate errors in the same way as throwing functions. For example, the initializer for the `PurchasedSnack` structure in the listing below calls a throwing function as part of the initialization process, and it handles any errors that it encounters by propagating them to its caller.

```
struct PurchasedSnack {
    let name: String
    init(name: String, vendingMachine:
     VendingMachine) throws {
        try vendingMachine.vend(itemNamed: name)
        self.name = name
    }
}
```

Handling Errors Using Do-Catch

You use a `do-catch` statement to handle errors by running a block of code. If an error is thrown by the code in the `do` clause, it's matched against the `catch` clauses to determine which one of them can handle the error.

Here is the general form of a `do-catch` statement:

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
} catch pattern 3, pattern 4 where condition {
    statements
} catch {
    statements
}
```

You write a pattern after `catch` to indicate what errors that clause can handle. If a `catch` clause doesn't have a pattern, the clause matches any error and binds the error to a local constant named `error`. For more information about pattern matching, see Patterns.

For example, the following code matches against all three cases of the `VendingMachineError` enumeration.

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice",
     vendingMachine: vendingMachine)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let
     coinsNeeded) {
    print("Insufficient funds. Please insert an
     additional \(coinsNeeded) coins.")
} catch {
    print("Unexpected error: \(error).")
}
// Prints "Insufficient funds. Please insert an
     additional 2 coins."
```

In the above example, the `buyFavoriteSnack(person:vendingMachine:)` function is called in a `try` expression, because it can throw an error. If an error is thrown, execution immediately transfers to the `catch` clauses, which decide whether to allow propagation to continue. If no pattern is matched, the error gets caught by the final `catch` clause and is bound to a local `error` constant. If no error is thrown, the remaining statements in the `do` statement are executed.

The `catch` clauses don't have to handle every possible error that the code in the `do` clause can throw. If none of the `catch` clauses handle the error, the error propagates to the surrounding scope. However, the propagated error must be handled by *some* surrounding scope. In a nonthrowing function, an enclosing `do-catch` statement must handle the error. In a throwing function, either an enclosing `do-catch` statement or the caller must handle the error. If the error propagates to the top-level scope without being handled, you'll get a runtime error.

For example, the above example can be written so any error that isn't a `VendingMachineError` is instead caught by the calling function:

```
func nourish(with item: String) throws {
    do {
        try vendingMachine.vend(itemNamed: item)
    } catch is VendingMachineError {
        print("Couldn't buy that from the vending machine.")
    }
}

do {
    try nourish(with: "Beet-Flavored Chips")
} catch {
    print("Unexpected non-vending-machine-related error: \(error)")
}
// Prints "Couldn't buy that from the vending machine."
```

In the `nourish(with:)` function, if `vend(itemNamed:)` throws an error that's one of the cases of the `VendingMachineError` enumeration, `nourish(with:)` handles the error by printing a message. Otherwise, `nourish(with:)` propagates the error to its call site. The error is then caught by the general `catch` clause.

Another way to catch several related errors is to list them after `catch`, separated by commas. For example:

```
func eat(item: String) throws {
    do {
        try vendingMachine.vend(itemNamed: item)
    } catch VendingMachineError.invalidSelection,
     VendingMachineError.insufficientFunds,
     VendingMachineError.outOfStock {
        print("Invalid selection, out of stock, or
     not enough money.")
    }
}
```

The `eat(item:)` function lists the vending machine errors to catch, and its error text corresponds to the items in that list. If any of the three listed errors are thrown, this `catch` clause handles them by printing a message. Any other errors are propagated to the surrounding scope, including any vending-machine errors that might be added later.

Converting Errors to Optional Values

You use `try?` to handle an error by converting it to an optional value. If an error is thrown while evaluating the `try?` expression, the value of the

expression is `nil`. For example, in the following code `x` and `y` have the same value and behavior:

```
func someThrowingFunction() throws -> Int {
    // ...
}

let x = try? someThrowingFunction()

let y: Int?
do {
    y = try someThrowingFunction()
} catch {
    y = nil
}
```

If `someThrowingFunction()` throws an error, the value of `x` and `y` is `nil`. Otherwise, the value of `x` and `y` is the value that the function returned. Note that `x` and `y` are an optional of whatever type `someThrowingFunction()` returns. Here the function returns an integer, so `x` and `y` are optional integers.

Using `try?` lets you write concise error handling code when you want to handle all errors in the same way. For example, the following code uses several approaches to fetch data, or returns `nil` if all of the approaches fail.

```
func fetchData() -> Data? {
    if let data = try? fetchDataFromDisk() { return data }
    if let data = try? fetchDataFromServer() { return data }
    return nil
}
```

Disabling Error Propagation
Sometimes you know a throwing function or method won't, in fact, throw an error at runtime. On those occasions, you can write `try!` before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, you'll get a runtime error.

For example, the following code uses a `loadImage(atPath:)` function, which loads the image resource at a given path or throws an error if the image can't be loaded. In this case, because the image is shipped with the application, no error will be thrown at runtime, so it's appropriate to disable error propagation.

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Specifying Cleanup Actions

You use a `defer` statement to execute a set of statements just before code execution leaves the current block of code. This statement lets you do any

necessary cleanup that should be performed regardless of *how* execution leaves the current block of code—whether it leaves because an error was thrown or because of a statement such as `return` or `break`. For example, you can use a `defer` statement to ensure that file descriptors are closed and manually allocated memory is freed.

A `defer` statement defers execution until the current scope is exited. This statement consists of the `defer` keyword and the statements to be executed later. The deferred statements may not contain any code that would transfer control out of the statements, such as a `break` or a `return` statement, or by throwing an error. Deferred actions are executed in the reverse of the order that they're written in your source code. That is, the code in the first `defer` statement executes last, the code in the second `defer` statement executes second to last, and so on. The last `defer` statement in source code order executes first.

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

The above example uses a `defer` statement to ensure that the `open(_:)` function has a corresponding call to `close(_:)`.

> NOTE
>
> You can use a `defer` statement even when no error handling code is involved.

Concurrency

Swift has built-in support for writing asynchronous and parallel code in a structured way. *Asynchronous code* can be suspended and resumed later, although only one piece of the program executes at a time. Suspending and resuming code in your program lets it continue to make progress on short-term operations like updating its UI while continuing to work on long-running operations like fetching data over the network or parsing files. *Parallel code* means multiple pieces of code run simultaneously—for example, a computer with a four-core processor can run four pieces of code at the same time, with each core carrying out one of the tasks. A program that uses parallel and asynchronous code carries out multiple operations at a time; it suspends operations that are waiting for an external system, and makes it easier to write this code in a memory-safe way.

The additional scheduling flexibility from parallel or asynchronous code also comes with a cost of increased complexity. Swift lets you express your intent in a way that enables some compile-time checking—for example, you can use actors to safely access mutable state. However, adding concurrency to slow or buggy code isn't a guarantee that it will become fast or correct. In fact, adding concurrency might even make your code harder to debug. However, using Swift's language-level support for concurrency in code that needs to be concurrent means Swift can help you catch problems at compile time.

The rest of this chapter uses the term *concurrency* to refer to this common combination of asynchronous and parallel code.

> NOTE
>
> If you've written concurrent code before, you might be used to working with threads. The concurrency model in Swift is built on top of threads, but you don't interact with them directly. An asynchronous function in Swift can give up the thread that it's running on, which lets another asynchronous function run on that thread while the first function is blocked.

Converted by Evan at Apps Dissected -www.appsdissected.com

Although it's possible to write concurrent code without using Swift's language support, that code tends to be harder to read. For example, the following code downloads a list of photo names, downloads the first photo in that list, and shows that photo to the user:

```
listPhotos(inGallery: "Summer Vacation") {
     photoNames in
    let sortedNames = photoNames.sorted()
    let name = sortedNames[0]
    downloadPhoto(named: name) { photo in
        show(photo)
    }
}
```

Even in this simple case, because the code has to be written as a series of completion handlers, you end up writing nested closures. In this style, more complex code with deep nesting can quickly become unwieldy.

Defining and Calling Asynchronous Functions

An *asynchronous function* or *asynchronous method* is a special kind of function or method that can be suspended while it's partway through execution. This is in contrast to ordinary, synchronous functions and methods, which either run to completion, throw an error, or never return. An asynchronous function or method still does one of those three things, but it can also pause in the middle when it's waiting for something. Inside the body of an asynchronous function or method, you mark each of these places where execution can be suspended.

To indicate that a function or method is asynchronous, you write the `async` keyword in its declaration after its parameters, similar to how you use `throws` to mark a throwing function. If the function or method returns a value, you write `async` before the return arrow (`->`). For example, here's how you might fetch the names of photos in a gallery:

```
1  func listPhotos(inGallery name: String) async ->
       [String] {
2      let result = // ... some asynchronous networking
       code ...
3      return result
4  }
```

For a function or method that's both asynchronous and throwing, you write `async` before `throws`.

When calling an asynchronous method, execution suspends until that method returns. You write `await` in front of the call to mark the possible suspension point. This is like writing `try` when calling a throwing function, to mark the possible change to the program's flow if there's an error. Inside an asynchronous method, the flow of execution is suspended *only* when you call another asynchronous method—suspension is never implicit or preemptive—which means every possible suspension point is marked with `await`.

For example, the code below fetches the names of all the pictures in a gallery and then shows the first picture:

```
1  let photoNames = await listPhotos(inGallery: "Summer
       Vacation")
2  let sortedNames = photoNames.sorted()
3  let name = sortedNames[0]
4  let photo = await downloadPhoto(named: name)
5  show(photo)
```

Because the `listPhotos(inGallery:)` and `downloadPhoto(named:)` functions both need to make network requests, they could take a relatively long time to complete. Making them both asynchronous by writing `async` before the return arrow lets the rest of the app's code keep running while this code waits for the picture to be ready.

To understand the concurrent nature of the example above, here's one possible order of execution:

1. The code starts running from the first line and runs up to the first `await`. It calls the `listPhotos(inGallery:)` function and suspends execution while it waits for that function to return.

2. While this code's execution is suspended, some other concurrent code in the same program runs. For example, maybe a long-running background task continues updating a list of new photo galleries. That code also runs until the next suspension point, marked by `await`, or until it completes.

3. After `listPhotos(inGallery:)` returns, this code continues execution starting at that point. It assigns the value that was returned to `photoNames`.

4. The lines that define `sortedNames` and `name` are regular, synchronous code. Because nothing is marked `await` on these lines, there aren't any possible suspension points.

5. The next `await` marks the call to the `downloadPhoto(named:)` function. This code pauses execution again until that function returns, giving other concurrent code an opportunity to run.

6. After `downloadPhoto(named:)` returns, its return value is assigned to `photo` and then passed as an argument when calling `show(_:)`.

The possible suspension points in your code marked with `await` indicate that the current piece of code might pause execution while waiting for the asynchronous function or method to return. This is also called *yielding the thread* because, behind the scenes, Swift suspends the execution of your code on the current thread and runs some other code on that thread instead. Because code with `await` needs to be able to suspend execution, only certain places in your program can call asynchronous functions or methods:

- Code in the body of an asynchronous function, method, or property.

- Code in the static `main()` method of a structure, class, or enumeration that's marked with `@main`.

- Code in a detached child task, as shown in Unstructured Concurrency below.

NOTE

The `Task.sleep(_:)` method is useful when writing simple code to learn how concurrency works. This method does nothing, but waits at least the given number of nanoseconds before it returns. Here's a version of the `listPhotos(inGallery:)` function that uses `sleep()` to simulate waiting for a network operation:

```
func listPhotos(inGallery name: String) async -> [String] {
    await Task.sleep(2 * 1_000_000_000)  // Two seconds
    return ["IMG001", "IMG99", "IMG0404"]
}
```

Asynchronous Sequences

The `listPhotos(inGallery:)` function in the previous section asynchronously returns the whole array at once, after all of the array's elements are ready. Another approach is to wait for one element of the collection at a time using an *asynchronous sequence*. Here's what iterating over an asynchronous sequence looks like:

```
import Foundation

let handle = FileHandle.standardInput
for try await line in handle.bytes.lines {
    print(line)
}
```

Instead of using an ordinary `for-in` loop, the example above writes `for` with `await` after it. Like when you call an asynchronous function or method, writing `await` indicates a possible suspension point. A `for-await-in` loop potentially suspends execution at the beginning of each iteration, when it's waiting for the next element to be available.

In the same way that you can use your own types in a `for-in` loop by adding conformance to the `Sequence` protocol, you can use your own types in a `for-await-in` loop by adding conformance to the `AsyncSequence` protocol.

Calling Asynchronous Functions in Parallel

Calling an asynchronous function with `await` runs only one piece of code at a time. While the asynchronous code is running, the caller waits for that code to finish before moving on to run the next line of code. For example, to fetch the first three photos from a gallery, you could await three calls to the `downloadPhoto(named:)` function as follows:

```
let firstPhoto = await downloadPhoto(named:
     photoNames[0])
let secondPhoto = await downloadPhoto(named:
     photoNames[1])
let thirdPhoto = await downloadPhoto(named:
     photoNames[2])

let photos = [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```

This approach has an important drawback: Although the download is asynchronous and lets other work happen while it progresses, only one call to `downloadPhoto(named:)` runs at a time. Each photo downloads completely before the next one starts downloading. However, there's no need for these operations to wait—each photo can download independently, or even at the same time.

To call an asynchronous function and let it run in parallel with code around it, write `async` in front of `let` when you define a constant, and then write `await` each time you use the constant.

```
1  async let firstPhoto = downloadPhoto(named:
       photoNames[0])
2  async let secondPhoto = downloadPhoto(named:
       photoNames[1])
3  async let thirdPhoto = downloadPhoto(named:
       photoNames[2])
4
5  let photos = await [firstPhoto, secondPhoto,
       thirdPhoto]
6  show(photos)
```

In this example, all three calls to `downloadPhoto(named:)` start without waiting for the previous one to complete. If there are enough system resources available, they can run at the same time. None of these function calls are marked with `await` because the code doesn't suspend to wait for the function's result. Instead, execution continues until the line where `photos` is defined—at that point, the program needs the results from these asynchronous calls, so you write `await` to pause execution until all three photos finish downloading.

Here's how you can think about the differences between these two approaches:

- Call asynchronous functions with `await` when the code on the following lines depends on that function's result. This creates work that is carried out sequentially.
- Call asynchronous functions with `async-let` when you don't need the result until later in your code. This creates work that can be carried out in parallel.
- Both `await` and `async-let` allow other code to run while they're suspended.
- In both cases, you mark the possible suspension point with `await` to indicate that execution will pause, if needed, until an asynchronous function has returned.

You can also mix both of these approaches in the same code.

Tasks and Task Groups

A *task* is a unit of work that can be run asynchronously as part of your program. All asynchronous code runs as part of some task. The `async-let` syntax described in the previous section creates a child task for you. You can also create a task group and add child tasks to that group, which gives you more control over priority and cancellation, and lets you create a dynamic number of tasks.

Tasks are arranged in a hierarchy. Each task in a task group has the same parent task, and each task can have child tasks. Because of the explicit relationship between tasks and task groups, this approach is called *structured concurrency*. Although you take on some of the responsibility for correctness, the explicit parent-child relationships between tasks lets Swift

handle some behaviors like propagating cancellation for you, and lets Swift detect some errors at compile time.

```
1  await withTaskGroup(of: Data.self) { taskGroup in
2      let photoNames = await listPhotos(inGallery:
        "Summer Vacation")
3      for name in photoNames {
4          taskGroup.addTask { await
        downloadPhoto(named: name) }
5      }
6  }
```

For more information about task groups, see `TaskGroup`.

Unstructured Concurrency

In addition to the structured approaches to concurrency described in the previous sections, Swift also supports unstructured concurrency. Unlike tasks that are part of a task group, an *unstructured task* doesn't have a parent task. You have complete flexibility to manage unstructured tasks in whatever way your program needs, but you're also completely responsible for their correctness. To create an unstructured task that runs on the current actor, call the `Task.init(priority:operation:)` initializer. To create an unstructured task that's not part of the current actor, known more specifically as a *detached task*, call the `Task.detached(priority:operation:)` class method. Both of these operations return a task handle that lets you interact with the task—for example, to wait for its result or to cancel it.

```swift
let newPhoto = // ... some photo data ...
let handle = Task {
    return await add(newPhoto, toGalleryNamed:
     "Spring Adventures")
}
let result = await handle.value
```

For more information about managing detached tasks, see `Task`.

Task Cancellation

Swift concurrency uses a cooperative cancellation model. Each task checks whether it has been canceled at the appropriate points in its execution, and responds to cancellation in whatever way is appropriate. Depending on the work you're doing, that usually means one of the following:

- Throwing an error like `CancellationError`
- Returning `nil` or an empty collection
- Returning the partially completed work

To check for cancellation, either call `Task.checkCancellation()`, which throws `CancellationError` if the task has been canceled, or check the value of `Task.isCancelled` and handle the cancellation in your own code. For example, a task that's downloading photos from a gallery might need to delete partial downloads and close network connections.

To propagate cancellation manually, call `Task.cancel()`.

Actors

Like classes, actors are reference types, so the comparison of value types and reference types in Classes Are Reference Types applies to actors as well as classes. Unlike classes, actors allow only one task to access their mutable state at a time, which makes it safe for code in multiple tasks to interact with the same instance of an actor. For example, here's an actor that records temperatures:

```
actor TemperatureLogger {
    let label: String
    var measurements: [Int]
    private(set) var max: Int

    init(label: String, measurement: Int) {
        self.label = label
        self.measurements = [measurement]
        self.max = measurement
    }
}
```

You introduce an actor with the `actor` keyword, followed by its definition in a pair of braces. The `TemperatureLogger` actor has properties that other code outside the actor can access, and restricts the `max` property so only code inside the actor can update the maximum value.

You create an instance of an actor using the same initializer syntax as structures and classes. When you access a property or method of an actor, you use `await` to mark the potential suspension point—for example:

```
let logger = TemperatureLogger(label: "Outdoors",
     measurement: 25)
print(await logger.max)
// Prints "25"
```

In this example, accessing `logger.max` is a possible suspension point. Because the actor allows only one task at a time to access its mutable state, if code from another task is already interacting with the logger, this code suspends while it waits to access the property.

In contrast, code that's part of the actor doesn't write `await` when accessing the actor's properties. For example, here's a method that updates a `TemperatureLogger` with a new temperature:

```
extension TemperatureLogger {
    func update(with measurement: Int) {
        measurements.append(measurement)
        if measurement > max {
            max = measurement
        }
    }
}
```

The `update(with:)` method is already running on the actor, so it doesn't mark its access to properties like `max` with `await`. This method also shows one of the reasons why actors allow only one task at a time to interact with their mutable state: Some updates to an actor's state temporarily break invariants. The `TemperatureLogger` actor keeps track of a list of temperatures and a maximum temperature, and it updates the maximum temperature when you record a new measurement. In the middle of an update, after appending the new measurement but before updating `max`, the

temperature logger is in a temporary inconsistent state. Preventing multiple tasks from interacting with the same instance simultaneously prevents problems like the following sequence of events:

1. Your code calls the `update(with:)` method. It updates the `measurements` array first.

2. Before your code can update `max`, code elsewhere reads the maximum value and the array of temperatures.

3. Your code finishes its update by changing `max`.

In this case, the code running elsewhere would read incorrect information because its access to the actor was interleaved in the middle of the call to `update(with:)` while the data was temporarily invalid. You can prevent this problem when using Swift actors because they only allow one operation on their state at a time, and because that code can be interrupted only in places where `await` marks a suspension point. Because `update(with:)` doesn't contain any suspension points, no other code can access the data in the middle of an update.

If you try to access those properties from outside the actor, like you would with an instance of a class, you'll get a compile-time error; for example:

```
print(logger.max)  // Error
```

Accessing `logger.max` without writing `await` fails because the properties of an actor are part of that actor's isolated local state. Swift guarantees that only code inside an actor can access the actor's local state. This guarantee is known as *actor isolation*.

Type Casting

Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.

Type casting in Swift is implemented with the `is` and `as` operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.

You can also use type casting to check whether a type conforms to a protocol, as described in Checking for Protocol Conformance.

Defining a Class Hierarchy for Type Casting

You can use type casting with a hierarchy of classes and subclasses to check the type of a particular class instance and to cast that instance to another class within the same hierarchy. The three code snippets below define a hierarchy of classes and an array containing instances of those classes, for use in an example of type casting.

The first snippet defines a new base class called `MediaItem`. This class provides basic functionality for any kind of item that appears in a digital media library. Specifically, it declares a `name` property of type `String`, and an `init name` initializer. (It's assumed that all media items, including all movies and songs, will have a name.)

```
class MediaItem {
    var name: String
    init(name: String) {
        self.name = name
    }
}
```

The next snippet defines two subclasses of `MediaItem`. The first subclass, `Movie`, encapsulates additional information about a movie or film. It adds a `director` property on top of the base `MediaItem` class, with a corresponding initializer. The second subclass, `Song`, adds an `artist` property and initializer on top of the base class:

```
class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

The final snippet creates a constant array called `library`, which contains two `Movie` instances and three `Song` instances. The type of the `library` array is inferred by initializing it with the contents of an array literal. Swift's type checker is able to deduce that `Movie` and `Song` have a common superclass of `MediaItem`, and so it infers a type of `[MediaItem]` for the `library` array:

```
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),
    Movie(name: "Citizen Kane", director: "Orson Welles"),
    Song(name: "The One And Only", artist: "Chesney Hawkes"),
    Song(name: "Never Gonna Give You Up", artist: "Rick Astley")
]
// the type of "library" is inferred to be [MediaItem]
```

The items stored in `library` are still `Movie` and `Song` instances behind the scenes. However, if you iterate over the contents of this array, the items you receive back are typed as `MediaItem`, and not as `Movie` or `Song`. In order to work with them as their native type, you need to *check* their type, or *downcast* them to a different type, as described below.

Checking Type

Use the *type check operator* (`is`) to check whether an instance is of a certain subclass type. The type check operator returns `true` if the instance is of that subclass type and `false` if it's not.

The example below defines two variables, `movieCount` and `songCount`, which count the number of `Movie` and `Song` instances in the `library` array:

```
var movieCount = 0
var songCount = 0

for item in library {
    if item is Movie {
        movieCount += 1
    } else if item is Song {
        songCount += 1
    }
}

print("Media library contains \(movieCount) movies and \(songCount) songs")
// Prints "Media library contains 2 movies and 3 songs"
```

This example iterates through all items in the `library` array. On each pass, the `for-in` loop sets the `item` constant to the next `MediaItem` in the array.

`item is Movie` returns `true` if the current `MediaItem` is a `Movie` instance and `false` if it's not. Similarly, `item is Song` checks whether the item is a `Song` instance. At the end of the `for-in` loop, the values of `movieCount` and `songCount` contain a count of how many `MediaItem` instances were found of each type.

Downcasting

A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to *downcast* to the subclass type with a *type cast operator* (`as?` or `as!`).

Because downcasting can fail, the type cast operator comes in two different forms. The conditional form, `as?`, returns an optional value of the type you are trying to downcast to. The forced form, `as!`, attempts the downcast and force-unwraps the result as a single compound action.

Use the conditional form of the type cast operator (`as?`) when you aren't sure if the downcast will succeed. This form of the operator will always return an optional value, and the value will be `nil` if the downcast was not possible. This enables you to check for a successful downcast.

Use the forced form of the type cast operator (`as!`) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type.

The example below iterates over each `MediaItem` in `library`, and prints an appropriate description for each item. To do this, it needs to access each item as a true `Movie` or `Song`, and not just as a `MediaItem`. This is necessary in order for it to be able to access the `director` or `artist` property of a `Movie` or `Song` for use in the description.

In this example, each item in the array might be a `Movie`, or it might be a `Song`. You don't know in advance which actual class to use for each item, and so it's appropriate to use the conditional form of the type cast operator (`as?`) to check the downcast each time through the loop:

```
for item in library {
    if let movie = item as? Movie {
        print("Movie: \(movie.name), dir. \(movie.director)")
    } else if let song = item as? Song {
        print("Song: \(song.name), by \(song.artist)")
    }
}

// Movie: Casablanca, dir. Michael Curtiz
// Song: Blue Suede Shoes, by Elvis Presley
// Movie: Citizen Kane, dir. Orson Welles
// Song: The One And Only, by Chesney Hawkes
// Song: Never Gonna Give You Up, by Rick Astley
```

The example starts by trying to downcast the current `item` as a `Movie`. Because `item` is a `MediaItem` instance, it's possible that it *might* be a `Movie`; equally, it's also possible that it might be a `Song`, or even just a base `MediaItem`. Because of this uncertainty, the `as?` form of the type cast operator returns an *optional* value when attempting to downcast to a subclass type. The result of `item as? Movie` is of type `Movie?`, or "optional `Movie`".

Downcasting to `Movie` fails when applied to the `Song` instances in the library array. To cope with this, the example above uses optional binding to check whether the optional `Movie` actually contains a value (that is, to find out whether the downcast succeeded.) This optional binding is written "`if let movie = item as? Movie`", which can be read as:

"Try to access `item` as a `Movie`. If this is successful, set a new temporary constant called `movie` to the value stored in the returned optional `Movie`."

If the downcasting succeeds, the properties of `movie` are then used to print a description for that `Movie` instance, including the name of its `director`. A similar principle is used to check for `Song` instances, and to print an appropriate description (including `artist` name) whenever a `Song` is found in the library.

> NOTE
>
> Casting doesn't actually modify the instance or change its values. The underlying instance remains the same; it's simply treated and accessed as an instance of the type to which it has been cast.

Type Casting for Any and AnyObject

Swift provides two special types for working with nonspecific types:

- `Any` can represent an instance of any type at all, including function types.
- `AnyObject` can represent an instance of any class type.

Use `Any` and `AnyObject` only when you explicitly need the behavior and capabilities they provide. It's always better to be specific about the types you expect to work with in your code.

Here's an example of using `Any` to work with a mix of different types, including function types and nonclass types. The example creates an array called `things`, which can store values of type `Any`:

```
var things: [Any] = []

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director:
     "Ivan Reitman"))
things.append({ (name: String) -> String in "Hello,
     \(name)" })
```

The `things` array contains two `Int` values, two `Double` values, a `String` value, a tuple of type `(Double, Double)`, the movie "Ghostbusters", and a closure expression that takes a `String` value and returns another `String` value.

To discover the specific type of a constant or variable that's known only to be of type `Any` or `AnyObject`, you can use an `is` or `as` pattern in a `switch` statement's cases. The example below iterates over the items in the `things` array and queries the type of each item with a `switch` statement. Several of the `switch` statement's cases bind their matched value to a constant of the specified type to enable its value to be printed:

```
for thing in things {
    switch thing {
    case 0 as Int:
        print("zero as an Int")
    case 0 as Double:
        print("zero as a Double")
    case let someInt as Int:
        print("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        print("a positive double value of \(someDouble)")
    case is Double:
        print("some other double value that I don't want to print")
    case let someString as String:
        print("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        print("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        print("a movie called \(movie.name), dir. \(movie.director)")
    case let stringConverter as (String) -> String:
        print(stringConverter("Michael"))
    default:
        print("something else")
```

```
    }
}

// zero as an Int
// zero as a Double
// an integer value of 42
// a positive double value of 3.14159
// a string value of "hello"
// an (x, y) point at 3.0, 5.0
// a movie called Ghostbusters, dir. Ivan Reitman
// Hello, Michael
```

NOTE

The `Any` type represents values of any type, including optional types. Swift gives you a warning if you use an optional value where a value of type `Any` is expected. If you really do need to use an optional value as an `Any` value, you can use the `as` operator to explicitly cast the optional to `Any`, as shown below.

```
let optionalNumber: Int? = 3
things.append(optionalNumber)        // Warning
things.append(optionalNumber as Any) // No warning
```

Nested Types

Enumerations are often created to support a specific class or structure's functionality. Similarly, it can be convenient to define utility classes and structures purely for use within the context of a more complex type. To accomplish this, Swift enables you to define *nested types*, whereby you nest supporting enumerations, classes, and structures within the definition of the type they support.

To nest a type within another type, write its definition within the outer braces of the type it supports. Types can be nested to as many levels as are required.

Nested Types in Action

The example below defines a structure called `BlackjackCard`, which models a playing card as used in the game of Blackjack. The `BlackjackCard` structure contains two nested enumeration types called `Suit` and `Rank`.

In Blackjack, the Ace cards have a value of either one or eleven. This feature is represented by a structure called `Values`, which is nested within the `Rank` enumeration:

```
struct BlackjackCard {

    // nested Suit enumeration
    enum Suit: Character {
        case spades = "♠", hearts = "♡", diamonds = "♢", clubs = "♣"
    }

    // nested Rank enumeration
    enum Rank: Int {
        case two = 2, three, four, five, six, seven, eight, nine, ten
        case jack, queen, king, ace
        struct Values {
            let first: Int, second: Int?
        }
        var values: Values {
            switch self {
            case .ace:
                return Values(first: 1, second: 11)
            case .jack, .queen, .king:
                return Values(first: 10, second: nil)
            default:
                return Values(first: self.rawValue, second: nil)
```

```
23                }
24            }
25        }
26
27        // BlackjackCard properties and methods
28        let rank: Rank, suit: Suit
29        var description: String {
30            var output = "suit is \(suit.rawValue),"
31            output += " value is \(rank.values.first)"
32            if let second = rank.values.second {
33                output += " or \(second)"
34            }
35            return output
36        }
37    }
```

The `Suit` enumeration describes the four common playing card suits, together with a raw `Character` value to represent their symbol.

The `Rank` enumeration describes the thirteen possible playing card ranks, together with a raw `Int` value to represent their face value. (This raw `Int` value isn't used for the Jack, Queen, King, and Ace cards.)

As mentioned above, the `Rank` enumeration defines a further nested structure of its own, called `Values`. This structure encapsulates the fact that most cards have one value, but the Ace card has two values. The `Values` structure defines two properties to represent this:

- `first`, of type `Int`

- `second`, of type `Int?`, or "optional `Int`"

`Rank` also defines a computed property, `values`, which returns an instance of the `Values` structure. This computed property considers the rank of the card and initializes a new `Values` instance with appropriate values based on its rank. It uses special values for `jack`, `queen`, `king`, and `ace`. For the numeric cards, it uses the rank's raw `Int` value.

The `BlackjackCard` structure itself has two properties — `rank` and `suit`. It also defines a computed property called `description`, which uses the values stored in `rank` and `suit` to build a description of the name and value of the card. The `description` property uses optional binding to check whether there's a second value to display, and if so, inserts additional description detail for that second value.

Because `BlackjackCard` is a structure with no custom initializers, it has an implicit memberwise initializer, as described in [Memberwise Initializers for Structure Types]. You can use this initializer to initialize a new constant called `theAceOfSpades`:

```
let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades)
print("theAceOfSpades: \(theAceOfSpades.description)")
// Prints "theAceOfSpades: suit is ♠, value is 1 or 11"
```

Even though `Rank` and `Suit` are nested within `BlackjackCard`, their type can be inferred from context, and so the initialization of this instance is able to refer to the enumeration cases by their case names (`.ace` and `.spades`) alone. In the example above, the `description` property correctly reports that the Ace of Spades has a value of `1` or `11`.

Referring to Nested Types

To use a nested type outside of its definition context, prefix its name with the name of the type it's nested within:

```
let heartsSymbol =
     BlackjackCard.Suit.hearts.rawValue
// heartsSymbol is "♡"
```

For the example above, this enables the names of `Suit`, `Rank`, and `Values` to be kept deliberately short, because their names are naturally qualified by the context in which they're defined.

Extensions

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you don't have access to the original source code (known as *retroactive modeling*). Extensions are similar to categories in Objective-C. (Unlike Objective-C categories, Swift extensions don't have names.)

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

In Swift, you can even extend a protocol to provide implementations of its requirements or add additional functionality that conforming types can take advantage of. For more details, see Protocol Extensions.

> NOTE
>
> Extensions can add new functionality to a type, but they can't override existing functionality.

Extension Syntax

Declare extensions with the `extension` keyword:

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

An extension can extend an existing type to make it adopt one or more protocols. To add protocol conformance, you write the protocol names the same way as you write them for a class or structure:

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

Adding protocol conformance in this way is described in Adding Protocol Conformance with an Extension.

An extension can be used to extend an existing generic type, as described in Extending a Generic Type. You can also extend a generic type to conditionally add functionality, as described in Extensions with a Generic Where Clause.

> NOTE
>
> If you define an extension to add new functionality to an existing type, the new functionality will be available on all existing instances of that type, even if they were created before the extension was defined.

Computed Properties

Extensions can add computed instance properties and computed type properties to existing types. This example adds five computed instance properties to Swift's built-in `Double` type, to provide basic support for working with distance units:

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// Prints "One inch is 0.0254 meters"
let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// Prints "Three feet is 0.914399970739201 meters"
```

These computed properties express that a `Double` value should be considered as a certain unit of length. Although they're implemented as computed properties, the names of these properties can be appended to a floating-point literal value with dot syntax, as a way to use that literal value to perform distance conversions.

In this example, a `Double` value of `1.0` is considered to represent "one meter". This is why the `m` computed property returns `self`—the expression `1.m` is considered to calculate a `Double` value of `1.0`.

Other units require some conversion to be expressed as a value measured in meters. One kilometer is the same as 1,000 meters, so the `km` computed property multiplies the value by `1_000.00` to convert into a number expressed in meters. Similarly, there are 3.28084 feet in a meter, and so the `ft` computed property divides the underlying `Double` value by `3.28084`, to convert it from feet to meters.

These properties are read-only computed properties, and so they're expressed without the `get` keyword, for brevity. Their return value is of type `Double`, and can be used within mathematical calculations wherever a `Double` is accepted:

```
let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
// Prints "A marathon is 42195.0 meters long"
```

> NOTE
>
> Extensions can add new computed properties, but they can't add stored properties, or add property observers to existing properties.

Initializers

Extensions can add new initializers to existing types. This enables you to extend other types to accept your own custom types as initializer parameters, or to provide additional initialization options that were not included as part of the type's original implementation.

Extensions can add new convenience initializers to a class, but they can't add new designated initializers or deinitializers to a class. Designated initializers and deinitializers must always be provided by the original class implementation.

If you use an extension to add an initializer to a value type that provides default values for all of its stored properties and doesn't define any custom initializers, you can call the default initializer and memberwise initializer for that value type from within your extension's initializer. This wouldn't be the case if you had written the initializer as part of the value type's original implementation, as described in Initializer Delegation for Value Types.

If you use an extension to add an initializer to a structure that was declared in another module, the new initializer can't access `self` until it calls an initializer from the defining module.

The example below defines a custom `Rect` structure to represent a geometric rectangle. The example also defines two supporting structures called `Size` and `Point`, both of which provide default values of `0.0` for all of their properties:

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}
```

Because the `Rect` structure provides default values for all of its properties, it receives a default initializer and a memberwise initializer automatically, as described in Default Initializers. These initializers can be used to create new `Rect` instances:

```
let defaultRect = Rect()
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
```

You can extend the `Rect` structure to provide an additional initializer that takes a specific center point and size:

```
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

This new initializer starts by calculating an appropriate origin point based on the provided `center` point and `size` value. The initializer then calls the structure's automatic memberwise initializer `init(origin:size:)`, which stores the new origin and size values in the appropriate properties:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

> NOTE
>
> If you provide a new initializer with an extension, you are still responsible for making sure that each instance is fully initialized once the initializer completes.

Methods

Extensions can add new instance methods and type methods to existing types. The following example adds a new instance method called `repetitions` to the `Int` type:

```
extension Int {
    func repetitions(task: () -> Void) {
        for _ in 0..<self {
            task()
        }
    }
}
```

The `repetitions(task:)` method takes a single argument of type `() -> Void`, which indicates a function that has no parameters and doesn't return a value.

After defining this extension, you can call the `repetitions(task:)` method on any integer to perform a task that many number of times:

```
3.repetitions {
    print("Hello!")
}
// Hello!
// Hello!
// Hello!
```

Mutating Instance Methods

Instance methods added with an extension can also modify (or *mutate*) the instance itself. Structure and enumeration methods that modify `self` or its properties must mark the instance method as `mutating`, just like mutating methods from an original implementation.

The example below adds a new mutating method called `square` to Swift's `Int` type, which squares the original value:

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// someInt is now 9
```

Subscripts

Extensions can add new subscripts to an existing type. This example adds an integer subscript to Swift's built-in `Int` type. This subscript `[n]` returns the decimal digit `n` places in from the right of the number:

- `123456789[0]` returns `9`
- `123456789[1]` returns `8`

…and so on:

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
        for _ in 0..<digitIndex {
            decimalBase *= 10
        }
        return (self / decimalBase) % 10
    }
}
746381295[0]
// returns 5
746381295[1]
// returns 9
746381295[2]
// returns 2
746381295[8]
// returns 7
```

If the `Int` value doesn't have enough digits for the requested index, the subscript implementation returns `0`, as if the number had been padded with zeros to the left:

```
746381295[9]
// returns 0, as if you had requested:
0746381295[9]
```

Nested Types

Extensions can add new nested types to existing classes, structures, and enumerations:

```
extension Int {
    enum Kind {
        case negative, zero, positive
    }
    var kind: Kind {
        switch self {
        case 0:
            return .zero
        case let x where x > 0:
            return .positive
        default:
            return .negative
        }
    }
}
```

This example adds a new nested enumeration to `Int`. This enumeration, called `Kind`, expresses the kind of number that a particular integer represents. Specifically, it expresses whether the number is negative, zero, or positive.

This example also adds a new computed instance property to `Int`, called `kind`, which returns the appropriate `Kind` enumeration case for that integer.

The nested enumeration can now be used with any `Int` value:

```swift
func printIntegerKinds(_ numbers: [Int]) {
    for number in numbers {
        switch number.kind {
        case .negative:
            print("- ", terminator: "")
        case .zero:
            print("0 ", terminator: "")
        case .positive:
            print("+ ", terminator: "")
        }
    }
    print("")
}
printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
// Prints "+ + - 0 - 0 + "
```

This function, `printIntegerKinds(_:)`, takes an input array of `Int` values and iterates over those values in turn. For each integer in the array, the function considers the `kind` computed property for that integer, and prints an appropriate description.

> NOTE
>
> `number.kind` is already known to be of type `Int.Kind`. Because of this, all of the `Int.Kind` case values can be written in shorthand form inside the `switch` statement, such as `.negative` rather than `Int.Kind.negative`.

Protocols

A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be *adopted* by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to *conform* to that protocol.

In addition to specifying requirements that conforming types must implement, you can extend a protocol to implement some of these requirements or to implement additional functionality that conforming types can take advantage of.

Protocol Syntax

You define protocols in a very similar way to classes, structures, and enumerations:

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Custom types state that they adopt a particular protocol by placing the protocol's name after the type's name, separated by a colon, as part of their definition. Multiple protocols can be listed, and are separated by commas:

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}
```

If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```

Property Requirements

A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn't specify whether the property should be a stored property or a computed property—it only specifies the required property name and type. The protocol also specifies whether each property must be gettable or gettable *and* settable.

If a protocol requires a property to be gettable and settable, that property requirement can't be fulfilled by a constant stored property or a read-only computed property. If the protocol only requires a property to be gettable, the requirement can be satisfied by any kind of property, and it's valid for the property to be also settable if this is useful for your own code.

Property requirements are always declared as variable properties, prefixed with the `var` keyword. Gettable and settable properties are indicated by

writing `{ get set }` after their type declaration, and gettable properties are indicated by writing `{ get }`.

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}
```

Always prefix type property requirements with the `static` keyword when you define them in a protocol. This rule pertains even though type property requirements can be prefixed with the `class` or `static` keyword when implemented by a class:

```
protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

Here's an example of a protocol with a single instance property requirement:

```
protocol FullyNamed {
    var fullName: String { get }
}
```

The `FullyNamed` protocol requires a conforming type to provide a fully qualified name. The protocol doesn't specify anything else about the nature of the conforming type—it only specifies that the type must be able to provide a full name for itself. The protocol states that any `FullyNamed` type must have a gettable instance property called `fullName`, which is of type `String`.

Here's an example of a simple structure that adopts and conforms to the `FullyNamed` protocol:

```
struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

This example defines a structure called `Person`, which represents a specific named person. It states that it adopts the `FullyNamed` protocol as part of the first line of its definition.

Each instance of `Person` has a single stored property called `fullName`, which is of type `String`. This matches the single requirement of the `FullyNamed` protocol, and means that `Person` has correctly conformed to the protocol. (Swift reports an error at compile time if a protocol requirement isn't fulfilled.)

Here's a more complex class, which also adopts and conforms to the `FullyNamed` protocol:

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName is "USS Enterprise"
```

This class implements the `fullName` property requirement as a computed read-only property for a starship. Each `Starship` class instance stores a mandatory `name` and an optional `prefix`. The `fullName` property uses the `prefix` value if it exists, and prepends it to the beginning of `name` to create a full name for the starship.

Method Requirements

Protocols can require specific instance methods and type methods to be implemented by conforming types. These methods are written as part of the protocol's definition in exactly the same way as for normal instance and

type methods, but without curly braces or a method body. Variadic parameters are allowed, subject to the same rules as for normal methods. Default values, however, can't be specified for method parameters within a protocol's definition.

As with type property requirements, you always prefix type method requirements with the `static` keyword when they're defined in a protocol. This is true even though type method requirements are prefixed with the `class` or `static` keyword when implemented by a class:

```
protocol SomeProtocol {
    static func someTypeMethod()
}
```

The following example defines a protocol with a single instance method requirement:

```
protocol RandomNumberGenerator {
    func random() -> Double
}
```

This protocol, `RandomNumberGenerator`, requires any conforming type to have an instance method called `random`, which returns a `Double` value whenever it's called. Although it's not specified as part of the protocol, it's assumed that this value will be a number from `0.0` up to (but not including) `1.0`.

The `RandomNumberGenerator` protocol doesn't make any assumptions about how each random number will be generated—it simply requires the generator to provide a standard way to generate a new random number.

Here's an implementation of a class that adopts and conforms to the `RandomNumberGenerator` protocol. This class implements a pseudorandom

number generator algorithm known as a *linear congruential generator*:

```
class LinearCongruentialGenerator:
     RandomNumberGenerator {
    var lastRandom = 42.0
    let m = 139968.0
    let a = 3877.0
    let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c)
            .truncatingRemainder(dividingBy:m))
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
print("Here's a random number: \
     (generator.random())")
// Prints "Here's a random number:
     0.3746499199817101"
print("And another one: \(generator.random())")
// Prints "And another one: 0.729023776863283"
```

Mutating Method Requirements

It's sometimes necessary for a method to modify (or *mutate*) the instance it belongs to. For instance methods on value types (that is, structures and enumerations) you place the `mutating` keyword before a method's `func` keyword to indicate that the method is allowed to modify the instance it belongs to and any properties of that instance. This process is described in [Modifying Value Types from Within Instance Methods](#).

If you define a protocol instance method requirement that's intended to mutate instances of any type that adopts the protocol, mark the method with the `mutating` keyword as part of the protocol's definition. This enables structures and enumerations to adopt the protocol and satisfy that method requirement.

> NOTE
>
> If you mark a protocol instance method requirement as `mutating`, you don't need to write the `mutating` keyword when writing an implementation of that method for a class. The `mutating` keyword is only used by structures and enumerations.

The example below defines a protocol called `Togglable`, which defines a single instance method requirement called `toggle`. As its name suggests, the `toggle()` method is intended to toggle or invert the state of any conforming type, typically by modifying a property of that type.

The `toggle()` method is marked with the `mutating` keyword as part of the `Togglable` protocol definition, to indicate that the method is expected to mutate the state of a conforming instance when it's called:

```
protocol Togglable {
    mutating func toggle()
}
```

If you implement the `Togglable` protocol for a structure or enumeration, that structure or enumeration can conform to the protocol by providing an implementation of the `toggle()` method that's also marked as `mutating`.

The example below defines an enumeration called `OnOffSwitch`. This enumeration toggles between two states, indicated by the enumeration cases `on` and `off`. The enumeration's `toggle` implementation is marked as `mutating`, to match the `Togglable` protocol's requirements:

```
enum OnOffSwitch: Togglable {
    case off, on
    mutating func toggle() {
        switch self {
        case .off:
            self = .on
        case .on:
            self = .off
        }
    }
}
var lightSwitch = OnOffSwitch.off
lightSwitch.toggle()
// lightSwitch is now equal to .on
```

Initializer Requirements

Protocols can require specific initializers to be implemented by conforming types. You write these initializers as part of the protocol's definition in exactly the same way as for normal initializers, but without curly braces or an initializer body:

```
protocol SomeProtocol {
    init(someParameter: Int)
}
```

Class Implementations of Protocol Initializer Requirements

You can implement a protocol initializer requirement on a conforming class as either a designated initializer or a convenience initializer. In both cases, you must mark the initializer implementation with the `required` modifier:

```
class SomeClass: SomeProtocol {
    required init(someParameter: Int) {
        // initializer implementation goes here
    }
}
```

The use of the `required` modifier ensures that you provide an explicit or inherited implementation of the initializer requirement on all subclasses of the conforming class, such that they also conform to the protocol.

For more information on required initializers, see Required Initializers.

> NOTE
>
> You don't need to mark protocol initializer implementations with the `required` modifier on classes that are marked with the `final` modifier, because final classes can't subclassed. For more about the `final` modifier, see Preventing Overrides.

If a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol, mark the initializer implementation with both the `required` and `override` modifiers:

```swift
protocol SomeProtocol {
    init()
}

class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from SomeSuperClass
    required override init() {
        // initializer implementation goes here
    }
}
```

Failable Initializer Requirements
Protocols can define failable initializer requirements for conforming types, as defined in Failable Initializers.

A failable initializer requirement can be satisfied by a failable or nonfailable initializer on a conforming type. A nonfailable initializer requirement can be satisfied by a nonfailable initializer or an implicitly unwrapped failable initializer.

Protocols as Types

Protocols don't actually implement any functionality themselves. Nonetheless, you can use protocols as a fully fledged types in your code. Using a protocol as a type is sometimes called an *existential type*, which comes from the phrase "there exists a type *T* such that *T* conforms to the protocol".

You can use a protocol in many places where other types are allowed, including:

- As a parameter type or return type in a function, method, or initializer
- As the type of a constant, variable, or property
- As the type of items in an array, dictionary, or other container

> NOTE
>
> Because protocols are types, begin their names with a capital letter (such as `FullyNamed` and `RandomNumberGenerator`) to match the names of other types in Swift (such as `Int`, `String`, and `Double`).

Here's an example of a protocol used as a type:

```
class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}
```

This example defines a new class called `Dice`, which represents an *n*-sided dice for use in a board game. `Dice` instances have an integer property called `sides`, which represents how many sides they have, and a property called `generator`, which provides a random number generator from which to create dice roll values.

The `generator` property is of type `RandomNumberGenerator`. Therefore, you can set it to an instance of *any* type that adopts the `RandomNumberGenerator` protocol. Nothing else is required of the instance you assign to this property, except that the instance must adopt the `RandomNumberGenerator` protocol. Because its type is `RandomNumberGenerator`, code inside the `Dice` class can only interact with `generator` in ways that apply to all generators that conform to this protocol. That means it can't use any methods or properties that are defined by the underlying type of the generator. However, you can downcast from a protocol type to an underlying type in the same way you can downcast from a superclass to a subclass, as discussed in Downcasting.

`Dice` also has an initializer, to set up its initial state. This initializer has a parameter called `generator`, which is also of type `RandomNumberGenerator`. You can pass a value of any conforming type in to this parameter when initializing a new `Dice` instance.

`Dice` provides one instance method, `roll`, which returns an integer value between 1 and the number of sides on the dice. This method calls the generator's `random()` method to create a new random number between `0.0` and `1.0`, and uses this random number to create a dice roll value within the correct range. Because `generator` is known to adopt `RandomNumberGenerator`, it's guaranteed to have a `random()` method to call.

Here's how the `Dice` class can be used to create a six-sided dice with a `LinearCongruentialGenerator` instance as its random number generator:

```
var d6 = Dice(sides: 6, generator:
     LinearCongruentialGenerator())
for _ in 1...5 {
    print("Random dice roll is \(d6.roll())")
}
// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// Random dice roll is 4
```

Delegation

Delegation is a design pattern that enables a class or structure to hand off (or *delegate*) some of its responsibilities to an instance of another type. This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated. Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

The example below defines two protocols for use with dice-based board games:

```
protocol DiceGame {
    var dice: Dice { get }
    func play()
}
protocol DiceGameDelegate: AnyObject {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame,
     didStartNewTurnWithDiceRoll diceRoll: Int)
    func gameDidEnd(_ game: DiceGame)
}
```

The `DiceGame` protocol is a protocol that can be adopted by any game that involves dice.

The `DiceGameDelegate` protocol can be adopted to track the progress of a `DiceGame`. To prevent strong reference cycles, delegates are declared as weak references. For information about weak references, see Strong Reference Cycles Between Class Instances. Marking the protocol as class-only lets the `SnakesAndLadders` class later in this chapter declare that its

delegate must use a weak reference. A class-only protocol is marked by its inheritance from `AnyObject`, as discussed in Class-Only Protocols.

Here's a version of the *Snakes and Ladders* game originally introduced in Control Flow. This version is adapted to use a `Dice` instance for its dice-rolls; to adopt the `DiceGame` protocol; and to notify a `DiceGameDelegate` about its progress:

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator: LinearCongruentialGenerator())
    var square = 0
    var board: [Int]
    init() {
        board = Array(repeating: 0, count: finalSquare + 1)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
    }
    weak var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll: diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
```

```
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

For a description of the *Snakes and Ladders* gameplay, see Break.

This version of the game is wrapped up as a class called `SnakesAndLadders`, which adopts the `DiceGame` protocol. It provides a gettable `dice` property and a `play()` method in order to conform to the protocol. (The `dice` property is declared as a constant property because it doesn't need to change after initialization, and the protocol only requires that it must be gettable.)

The *Snakes and Ladders* game board setup takes place within the class's `init()` initializer. All game logic is moved into the protocol's `play` method, which uses the protocol's required `dice` property to provide its dice roll values.

Note that the `delegate` property is defined as an *optional* `DiceGameDelegate`, because a delegate isn't required in order to play the game. Because it's of an optional type, the `delegate` property is automatically set to an initial value of `nil`. Thereafter, the game instantiator has the option to set the property to a suitable delegate. Because the

`DiceGameDelegate` protocol is class-only, you can declare the delegate to be `weak` to prevent reference cycles.

`DiceGameDelegate` provides three methods for tracking the progress of a game. These three methods have been incorporated into the game logic within the `play()` method above, and are called when a new game starts, a new turn begins, or the game ends.

Because the `delegate` property is an *optional* `DiceGameDelegate`, the `play()` method uses optional chaining each time it calls a method on the delegate. If the `delegate` property is nil, these delegate calls fail gracefully and without error. If the `delegate` property is non-nil, the delegate methods are called, and are passed the `SnakesAndLadders` instance as a parameter.

This next example shows a class called `DiceGameTracker`, which adopts the `DiceGameDelegate` protocol:

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(_ game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            print("Started a new game of Snakes and Ladders")
        }
        print("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll diceRoll: Int) {
        numberOfTurns += 1
        print("Rolled a \(diceRoll)")
    }
    func gameDidEnd(_ game: DiceGame) {
        print("The game lasted for \(numberOfTurns) turns")
    }
}
```

`DiceGameTracker` implements all three methods required by `DiceGameDelegate`. It uses these methods to keep track of the number of turns a game has taken. It resets a `numberOfTurns` property to zero when the game starts, increments it each time a new turn begins, and prints out the total number of turns once the game has ended.

The implementation of `gameDidStart(_:)` shown above uses the `game` parameter to print some introductory information about the game that's about to be played. The `game` parameter has a type of `DiceGame`, not `SnakesAndLadders`, and so `gameDidStart(_:)` can access and use only methods and properties that are implemented as part of the `DiceGame` protocol. However, the method is still able to use type casting to query the type of the underlying instance. In this example, it checks whether `game` is actually an instance of `SnakesAndLadders` behind the scenes, and prints an appropriate message if so.

The `gameDidStart(_:)` method also accesses the `dice` property of the passed `game` parameter. Because `game` is known to conform to the `DiceGame` protocol, it's guaranteed to have a `dice` property, and so the `gameDidStart(_:)` method is able to access and print the dice's `sides` property, regardless of what kind of game is being played.

Here's how `DiceGameTracker` looks in action:

```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Started a new game of Snakes and Ladders
// The game is using a 6-sided dice
// Rolled a 3
// Rolled a 5
// Rolled a 4
// Rolled a 5
// The game lasted for 4 turns
```

Adding Protocol Conformance with an Extension

You can extend an existing type to adopt and conform to a new protocol, even if you don't have access to the source code for the existing type. Extensions can add new properties, methods, and subscripts to an existing type, and are therefore able to add any requirements that a protocol may demand. For more about extensions, see Extensions.

> NOTE
>
> Existing instances of a type automatically adopt and conform to a protocol when that conformance is added to the instance's type in an extension.

For example, this protocol, called `TextRepresentable`, can be implemented by any type that has a way to be represented as text. This might be a description of itself, or a text version of its current state:

```
protocol TextRepresentable {
    var textualDescription: String { get }
}
```

The `Dice` class from above can be extended to adopt and conform to `TextRepresentable`:

```
extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}
```

This extension adopts the new protocol in exactly the same way as if `Dice` had provided it in its original implementation. The protocol name is

provided after the type name, separated by a colon, and an implementation of all requirements of the protocol is provided within the extension's curly braces.

Any `Dice` instance can now be treated as `TextRepresentable`:

```
let d12 = Dice(sides: 12, generator:
     LinearCongruentialGenerator())
print(d12.textualDescription)
// Prints "A 12-sided dice"
```

Similarly, the `SnakesAndLadders` game class can be extended to adopt and conform to the `TextRepresentable` protocol:

```
extension SnakesAndLadders: TextRepresentable {
    var textualDescription: String {
        return "A game of Snakes and Ladders with \
     (finalSquare) squares"
    }
}
print(game.textualDescription)
// Prints "A game of Snakes and Ladders with 25
     squares"
```

Conditionally Conforming to a Protocol

A generic type may be able to satisfy the requirements of a protocol only under certain conditions, such as when the type's generic parameter conforms to the protocol. You can make a generic type conditionally conform to a protocol by listing constraints when extending the type. Write

these constraints after the name of the protocol you're adopting by writing a generic `where` clause. For more about generic `where` clauses, see [Generic Where Clauses].

The following extension makes `Array` instances conform to the `TextRepresentable` protocol whenever they store elements of a type that conforms to `TextRepresentable`.

```
extension Array: TextRepresentable where Element: TextRepresentable {
    var textualDescription: String {
        let itemsAsText = self.map { $0.textualDescription }
        return "[" + itemsAsText.joined(separator: ", ") + "]"
    }
}
let myDice = [d6, d12]
print(myDice.textualDescription)
// Prints "[A 6-sided dice, A 12-sided dice]"
```

Declaring Protocol Adoption with an Extension

If a type already conforms to all of the requirements of a protocol, but hasn't yet stated that it adopts that protocol, you can make it adopt the protocol with an empty extension:

```
struct Hamster {
    var name: String
    var textualDescription: String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

Instances of `Hamster` can now be used wherever `TextRepresentable` is the required type:

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable =
     simonTheHamster
print(somethingTextRepresentable.textualDescription)
// Prints "A hamster named Simon"
```

> NOTE
>
> Types don't automatically adopt a protocol just by satisfying its requirements. They must always explicitly declare their adoption of the protocol.

Adopting a Protocol Using a Synthesized Implementation

Swift can automatically provide the protocol conformance for `Equatable`, `Hashable`, and `Comparable` in many simple cases. Using this synthesized

implementation means you don't have to write repetitive boilerplate code to implement the protocol requirements yourself.

Swift provides a synthesized implementation of `Equatable` for the following kinds of custom types:

- Structures that have only stored properties that conform to the `Equatable` protocol
- Enumerations that have only associated types that conform to the `Equatable` protocol
- Enumerations that have no associated types

To receive a synthesized implementation of `==`, declare conformance to `Equatable` in the file that contains the original declaration, without implementing an `==` operator yourself. The `Equatable` protocol provides a default implementation of `!=`.

The example below defines a `Vector3D` structure for a three-dimensional position vector `(x, y, z)`, similar to the `Vector2D` structure. Because the `x`, `y`, and `z` properties are all of an `Equatable` type, `Vector3D` receives synthesized implementations of the equivalence operators.

```swift
struct Vector3D: Equatable {
    var x = 0.0, y = 0.0, z = 0.0
}

let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0,
     z: 4.0)
if twoThreeFour == anotherTwoThreeFour {
    print("These two vectors are also equivalent.")
}
// Prints "These two vectors are also equivalent."
```

Swift provides a synthesized implementation of `Hashable` for the following kinds of custom types:

- Structures that have only stored properties that conform to the `Hashable` protocol
- Enumerations that have only associated types that conform to the `Hashable` protocol
- Enumerations that have no associated types

To receive a synthesized implementation of `hash(into:)`, declare conformance to `Hashable` in the file that contains the original declaration, without implementing a `hash(into:)` method yourself.

Swift provides a synthesized implementation of `Comparable` for enumerations that don't have a raw value. If the enumeration has associated types, they must all conform to the `Comparable` protocol. To receive a synthesized implementation of `<`, declare conformance to `Comparable` in the file that contains the original enumeration declaration, without

implementing a `<` operator yourself. The `Comparable` protocol's default implementation of `<=`, `>`, and `>=` provides the remaining comparison operators.

The example below defines a `SkillLevel` enumeration with cases for beginners, intermediates, and experts. Experts are additionally ranked by the number of stars they have.

```
enum SkillLevel: Comparable {
    case beginner
    case intermediate
    case expert(stars: Int)
}
var levels = [SkillLevel.intermediate,
     SkillLevel.beginner,
              SkillLevel.expert(stars: 5),
     SkillLevel.expert(stars: 3)]
for level in levels.sorted() {
    print(level)
}
// Prints "beginner"
// Prints "intermediate"
// Prints "expert(stars: 3)"
// Prints "expert(stars: 5)"
```

Collections of Protocol Types

A protocol can be used as the type to be stored in a collection such as an array or a dictionary, as mentioned in Protocols as Types. This example creates an array of `TextRepresentable` things:

```
let things: [TextRepresentable] = [game, d12,
     simonTheHamster]
```

It's now possible to iterate over the items in the array, and print each item's textual description:

```
for thing in things {
    print(thing.textualDescription)
}
// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

Note that the `thing` constant is of type `TextRepresentable`. It's not of type `Dice`, or `DiceGame`, or `Hamster`, even if the actual instance behind the scenes is of one of those types. Nonetheless, because it's of type `TextRepresentable`, and anything that's `TextRepresentable` is known to have a `textualDescription` property, it's safe to access `thing.textualDescription` each time through the loop.

Protocol Inheritance

A protocol can *inherit* one or more other protocols and can add further requirements on top of the requirements it inherits. The syntax for protocol

inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

```
protocol InheritingProtocol: SomeProtocol,
     AnotherProtocol {
    // protocol definition goes here
}
```

Here's an example of a protocol that inherits the `TextRepresentable` protocol from above:

```
protocol PrettyTextRepresentable: TextRepresentable
     {
    var prettyTextualDescription: String { get }
}
```

This example defines a new protocol, `PrettyTextRepresentable`, which inherits from `TextRepresentable`. Anything that adopts `PrettyTextRepresentable` must satisfy all of the requirements enforced by `TextRepresentable`, *plus* the additional requirements enforced by `PrettyTextRepresentable`. In this example, `PrettyTextRepresentable` adds a single requirement to provide a gettable property called `prettyTextualDescription` that returns a `String`.

The `SnakesAndLadders` class can be extended to adopt and conform to `PrettyTextRepresentable`:

```
extension SnakesAndLadders: PrettyTextRepresentable {
    var prettyTextualDescription: String {
        var output = textualDescription + ":\n"
        for index in 1...finalSquare {
            switch board[index] {
            case let ladder where ladder > 0:
                output += "▲ "
            case let snake where snake < 0:
                output += "▼ "
            default:
                output += "○ "
            }
        }
        return output
    }
}
```

This extension states that it adopts the `PrettyTextRepresentable` protocol and provides an implementation of the `prettyTextualDescription` property for the `SnakesAndLadders` type. Anything that's `PrettyTextRepresentable` must also be `TextRepresentable`, and so the implementation of `prettyTextualDescription` starts by accessing the `textualDescription` property from the `TextRepresentable` protocol to begin an output string. It appends a colon and a line break, and uses this as the start of its pretty text representation. It then iterates through the array of board squares, and appends a geometric shape to represent the contents of each square:

- If the square's value is greater than `0`, it's the base of a ladder, and is represented by ▲.
- If the square's value is less than `0`, it's the head of a snake, and is represented by ▼.
- Otherwise, the square's value is `0`, and it's a "free" square, represented by ○.

The `prettyTextualDescription` property can now be used to print a pretty text description of any `SnakesAndLadders` instance:

```
print(game.prettyTextualDescription)
// A game of Snakes and Ladders with 25 squares:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

Class-Only Protocols

You can limit protocol adoption to class types (and not structures or enumerations) by adding the `AnyObject` protocol to a protocol's inheritance list.

```
protocol SomeClassOnlyProtocol: AnyObject,
     SomeInheritedProtocol {
    // class-only protocol definition goes here
}
```

In the example above, `SomeClassOnlyProtocol` can only be adopted by class types. It's a compile-time error to write a structure or enumeration definition that tries to adopt `SomeClassOnlyProtocol`.

> NOTE
>
> Use a class-only protocol when the behavior defined by that protocol's requirements assumes or requires that a conforming type has reference semantics rather than value semantics. For more about reference and value semantics, see Structures and Enumerations Are Value Types and Classes Are Reference Types.

Protocol Composition

It can be useful to require a type to conform to multiple protocols at the same time. You can combine multiple protocols into a single requirement with a *protocol composition*. Protocol compositions behave as if you defined a temporary local protocol that has the combined requirements of all protocols in the composition. Protocol compositions don't define any new protocol types.

Protocol compositions have the form `SomeProtocol & AnotherProtocol`. You can list as many protocols as you need, separating them with ampersands (`&`). In addition to its list of protocols, a protocol composition can also contain one class type, which you can use to specify a required superclass.

Here's an example that combines two protocols called `Named` and `Aged` into a single protocol composition requirement on a function parameter:

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(to celebrator: Named & Aged) {
    print("Happy birthday, \(celebrator.name), you're \(celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(to: birthdayPerson)
// Prints "Happy birthday, Malcolm, you're 21!"
```

In this example, the `Named` protocol has a single requirement for a gettable `String` property called `name`. The `Aged` protocol has a single requirement for a gettable `Int` property called `age`. Both protocols are adopted by a structure called `Person`.

The example also defines a `wishHappyBirthday(to:)` function. The type of the `celebrator` parameter is `Named & Aged`, which means "any type that conforms to both the `Named` and `Aged` protocols." It doesn't matter

which specific type is passed to the function, as long as it conforms to both of the required protocols.

The example then creates a new `Person` instance called `birthdayPerson` and passes this new instance to the `wishHappyBirthday(to:)` function. Because `Person` conforms to both protocols, this call is valid, and the `wishHappyBirthday(to:)` function can print its birthday greeting.

Here's an example that combines the `Named` protocol from the previous example with a `Location` class:

```
class Location {
    var latitude: Double
    var longitude: Double
    init(latitude: Double, longitude: Double) {
        self.latitude = latitude
        self.longitude = longitude
    }
}
class City: Location, Named {
    var name: String
    init(name: String, latitude: Double, longitude: Double) {
        self.name = name
        super.init(latitude: latitude, longitude: longitude)
    }
}
func beginConcert(in location: Location & Named) {
    print("Hello, \(location.name)!")
}

let seattle = City(name: "Seattle", latitude: 47.6, longitude: -122.3)
beginConcert(in: seattle)
// Prints "Hello, Seattle!"
```

The `beginConcert(in:)` function takes a parameter of type `Location & Named`, which means "any type that's a subclass of `Location` and that conforms to the `Named` protocol." In this case, `City` satisfies both requirements.

Passing `birthdayPerson` to the `beginConcert(in:)` function is invalid because `Person` isn't a subclass of `Location`. Likewise, if you made a subclass of `Location` that didn't conform to the `Named` protocol, calling `beginConcert(in:)` with an instance of that type is also invalid.

Checking for Protocol Conformance

You can use the `is` and `as` operators described in Type Casting to check for protocol conformance, and to cast to a specific protocol. Checking for and casting to a protocol follows exactly the same syntax as checking for and casting to a type:

- The `is` operator returns `true` if an instance conforms to a protocol and returns `false` if it doesn't.
- The `as?` version of the downcast operator returns an optional value of the protocol's type, and this value is `nil` if the instance doesn't conform to that protocol.
- The `as!` version of the downcast operator forces the downcast to the protocol type and triggers a runtime error if the downcast doesn't succeed.

This example defines a protocol called `HasArea`, with a single property requirement of a gettable `Double` property called `area`:

```
protocol HasArea {
    var area: Double { get }
}
```

Here are two classes, `Circle` and `Country`, both of which conform to the `HasArea` protocol:

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

The `Circle` class implements the `area` property requirement as a computed property, based on a stored `radius` property. The `Country` class implements the `area` requirement directly as a stored property. Both classes correctly conform to the `HasArea` protocol.

Here's a class called `Animal`, which doesn't conform to the `HasArea` protocol:

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

The `Circle`, `Country` and `Animal` classes don't have a shared base class. Nonetheless, they're all classes, and so instances of all three types can be used to initialize an array that stores values of type `AnyObject`:

```
let objects: [AnyObject] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]
```

The `objects` array is initialized with an array literal containing a `Circle` instance with a radius of 2 units; a `Country` instance initialized with the surface area of the United Kingdom in square kilometers; and an `Animal` instance with four legs.

The `objects` array can now be iterated, and each object in the array can be checked to see if it conforms to the `HasArea` protocol:

```
for object in objects {
    if let objectWithArea = object as? HasArea {
        print("Area is \(objectWithArea.area)")
    } else {
        print("Something that doesn't have an area")
    }
}
// Area is 12.5663708
// Area is 243610.0
// Something that doesn't have an area
```

Whenever an object in the array conforms to the `HasArea` protocol, the optional value returned by the `as?` operator is unwrapped with optional binding into a constant called `objectWithArea`. The `objectWithArea` constant is known to be of type `HasArea`, and so its `area` property can be accessed and printed in a type-safe way.

Note that the underlying objects aren't changed by the casting process. They continue to be a `Circle`, a `Country` and an `Animal`. However, at the point that they're stored in the `objectWithArea` constant, they're only known to be of type `HasArea`, and so only their `area` property can be accessed.

Optional Protocol Requirements

You can define *optional requirements* for protocols. These requirements don't have to be implemented by types that conform to the protocol. Optional requirements are prefixed by the `optional` modifier as part of the protocol's definition. Optional requirements are available so that you can

write code that interoperates with Objective-C. Both the protocol and the optional requirement must be marked with the `@objc` attribute. Note that `@objc` protocols can be adopted only by classes that inherit from Objective-C classes or other `@objc` classes. They can't be adopted by structures or enumerations.

When you use a method or property in an optional requirement, its type automatically becomes an optional. For example, a method of type `(Int) -> String` becomes `((Int) -> String)?`. Note that the entire function type is wrapped in the optional, not the method's return value.

An optional protocol requirement can be called with optional chaining, to account for the possibility that the requirement was not implemented by a type that conforms to the protocol. You check for an implementation of an optional method by writing a question mark after the name of the method when it's called, such as `someOptionalMethod?(someArgument)`. For information on optional chaining, see Optional Chaining.

The following example defines an integer-counting class called `Counter`, which uses an external data source to provide its increment amount. This data source is defined by the `CounterDataSource` protocol, which has two optional requirements:

```
@objc protocol CounterDataSource {
    @objc optional func increment(forCount count: Int) -> Int
    @objc optional var fixedIncrement: Int { get }
}
```

The `CounterDataSource` protocol defines an optional method requirement called `increment(forCount:)` and an optional property requirement called `fixedIncrement`. These requirements define two different ways for data sources to provide an appropriate increment amount for a `Counter` instance.

> NOTE
>
> Strictly speaking, you can write a custom class that conforms to `CounterDataSource` without implementing *either* protocol requirement. They're both optional, after all. Although technically allowed, this wouldn't make for a very good data source.

The `Counter` class, defined below, has an optional `dataSource` property of type `CounterDataSource?`:

```
class Counter {
    var count = 0
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.increment?(forCount: count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement {
            count += amount
        }
    }
}
```

The `Counter` class stores its current value in a variable property called `count`. The `Counter` class also defines a method called `increment`, which increments the `count` property every time the method is called.

The `increment()` method first tries to retrieve an increment amount by looking for an implementation of the `increment(forCount:)` method on its data source. The `increment()` method uses optional chaining to try to

call `increment(forCount:)`, and passes the current `count` value as the method's single argument.

Note that *two* levels of optional chaining are at play here. First, it's possible that `dataSource` may be `nil`, and so `dataSource` has a question mark after its name to indicate that `increment(forCount:)` should be called only if `dataSource` isn't `nil`. Second, even if `dataSource` *does* exist, there's no guarantee that it implements `increment(forCount:)`, because it's an optional requirement. Here, the possibility that `increment(forCount:)` might not be implemented is also handled by optional chaining. The call to `increment(forCount:)` happens only if `increment(forCount:)` exists—that is, if it isn't `nil`. This is why `increment(forCount:)` is also written with a question mark after its name.

Because the call to `increment(forCount:)` can fail for either of these two reasons, the call returns an *optional* `Int` value. This is true even though `increment(forCount:)` is defined as returning a non-optional `Int` value in the definition of `CounterDataSource`. Even though there are two optional chaining operations, one after another, the result is still wrapped in a single optional. For more information about using multiple optional chaining operations, see Linking Multiple Levels of Chaining.

After calling `increment(forCount:)`, the optional `Int` that it returns is unwrapped into a constant called `amount`, using optional binding. If the optional `Int` does contain a value—that is, if the delegate and method both exist, and the method returned a value—the unwrapped `amount` is added onto the stored `count` property, and incrementation is complete.

If it's *not* possible to retrieve a value from the `increment(forCount:)` method—either because `dataSource` is nil, or because the data source doesn't implement `increment(forCount:)`—then the `increment()` method tries to retrieve a value from the data source's `fixedIncrement` property instead. The `fixedIncrement` property is also an optional requirement, so its value is an optional `Int` value, even though `fixedIncrement` is defined as a non-optional `Int` property as part of the `CounterDataSource` protocol definition.

Here's a simple `CounterDataSource` implementation where the data source returns a constant value of `3` every time it's queried. It does this by implementing the optional `fixedIncrement` property requirement:

```
class ThreeSource: NSObject, CounterDataSource {
    let fixedIncrement = 3
}
```

You can use an instance of `ThreeSource` as the data source for a new `Counter` instance:

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    print(counter.count)
}
// 3
// 6
// 9
// 12
```

The code above creates a new `Counter` instance; sets its data source to be a new `ThreeSource` instance; and calls the counter's `increment()` method four times. As expected, the counter's `count` property increases by three each time `increment()` is called.

Here's a more complex data source called `TowardsZeroSource`, which makes a `Counter` instance count up or down towards zero from its current `count` value:

```
class TowardsZeroSource: NSObject, CounterDataSource {
    func increment(forCount count: Int) -> Int {
        if count == 0 {
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```

The `TowardsZeroSource` class implements the optional `increment(forCount:)` method from the `CounterDataSource` protocol and uses the `count` argument value to work out which direction to count in. If `count` is already zero, the method returns `0` to indicate that no further counting should take place.

You can use an instance of `TowardsZeroSource` with the existing `Counter` instance to count from `-4` to zero. Once the counter reaches zero, no more counting takes place:

```
counter.count = -4
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    print(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

Protocol Extensions

Protocols can be extended to provide method, initializer, subscript, and computed property implementations to conforming types. This allows you to define behavior on protocols themselves, rather than in each type's individual conformance or in a global function.

For example, the `RandomNumberGenerator` protocol can be extended to provide a `randomBool()` method, which uses the result of the required `random()` method to return a random `Bool` value:

```
extension RandomNumberGenerator {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}
```

By creating an extension on the protocol, all conforming types automatically gain this method implementation without any additional modification.

```
let generator = LinearCongruentialGenerator()
print("Here's a random number: \
     (generator.random())")
// Prints "Here's a random number:
     0.3746499199817101"
print("And here's a random Boolean: \
     (generator.randomBool())")
// Prints "And here's a random Boolean: true"
```

Protocol extensions can add implementations to conforming types but can't make a protocol extend or inherit from another protocol. Protocol inheritance is always specified in the protocol declaration itself.

Providing Default Implementations

You can use protocol extensions to provide a default implementation to any method or computed property requirement of that protocol. If a conforming type provides its own implementation of a required method or property, that implementation will be used instead of the one provided by the extension.

> NOTE
>
> Protocol requirements with default implementations provided by extensions are distinct from optional protocol requirements. Although conforming types don't have to provide their own implementation of either, requirements with default implementations can be called without optional chaining.

For example, the `PrettyTextRepresentable` protocol, which inherits the `TextRepresentable` protocol can provide a default implementation of its required `prettyTextualDescription` property to simply return the result of accessing the `textualDescription` property:

```
extension PrettyTextRepresentable  {
    var prettyTextualDescription: String {
        return textualDescription
    }
}
```

Adding Constraints to Protocol Extensions

When you define a protocol extension, you can specify constraints that conforming types must satisfy before the methods and properties of the extension are available. You write these constraints after the name of the protocol you're extending by writing a generic `where` clause. For more about generic `where` clauses, see Generic Where Clauses.

For example, you can define an extension to the `Collection` protocol that applies to any collection whose elements conform to the `Equatable` protocol. By constraining a collection's elements to the `Equatable` protocol, a part of the standard library, you can use the `==` and `!=` operators to check for equality and inequality between two elements.

```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

The `allEqual()` method returns `true` only if all the elements in the collection are equal.

Consider two arrays of integers, one where all the elements are the same, and one where they aren't:

```
let equalNumbers = [100, 100, 100, 100, 100]
let differentNumbers = [100, 100, 200, 100, 200]
```

Because arrays conform to `Collection` and integers conform to `Equatable`, `equalNumbers` and `differentNumbers` can use the `allEqual()` method:

```
print(equalNumbers.allEqual())
// Prints "true"
print(differentNumbers.allEqual())
// Prints "false"
```

> NOTE
>
> If a conforming type satisfies the requirements for multiple constrained extensions that provide implementations for the same method or property, Swift uses the implementation corresponding to the most specialized constraints.

Converted by Evan at Apps Dissected -www.appsdissected.com

Generics

Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

Generics are one of the most powerful features of Swift, and much of the Swift standard library is built with generic code. In fact, you've been using generics throughout the *Language Guide*, even if you didn't realize it. For example, Swift's `Array` and `Dictionary` types are both generic collections. You can create an array that holds `Int` values, or an array that holds `String` values, or indeed an array for any other type that can be created in Swift. Similarly, you can create a dictionary to store values of any specified type, and there are no limitations on what that type can be.

The Problem That Generics Solve

Here's a standard, nongeneric function called `swapTwoInts(_:_:)`, which swaps two `Int` values:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

This function makes use of in-out parameters to swap the values of `a` and `b`, as described in [In-Out Parameters](#).

The `swapTwoInts(_:_:)` function swaps the original value of `b` into `a`, and the original value of `a` into `b`. You can call this function to swap the values in two `Int` variables:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

The `swapTwoInts(_:_:)` function is useful, but it can only be used with `Int` values. If you want to swap two `String` values, or two `Double` values, you have to write more functions, such as the `swapTwoStrings(_:_:)` and `swapTwoDoubles(_:_:)` functions shown below:

```
func swapTwoStrings(_ a: inout String, _ b: inout String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

You may have noticed that the bodies of the `swapTwoInts(_:_:)`, `swapTwoStrings(_:_:)`, and `swapTwoDoubles(_:_:)` functions are identical. The only difference is the type of the values that they accept (`Int`, `String`, and `Double`).

It's more useful, and considerably more flexible, to write a single function that swaps two values of *any* type. Generic code enables you to write such a function. (A generic version of these functions is defined below.)

> NOTE
>
> In all three functions, the types of `a` and `b` must be the same. If `a` and `b` aren't of the same type, it isn't possible to swap their values. Swift is a type-safe language, and doesn't allow (for example) a variable of type `String` and a variable of type `Double` to swap values with each other. Attempting to do so results in a compile-time error.

Generic Functions

Generic functions can work with any type. Here's a generic version of the `swapTwoInts(_:_:)` function from above, called `swapTwoValues(_:_:)`:

```
1  func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
2      let temporaryA = a
3      a = b
4      b = temporaryA
5  }
```

The body of the `swapTwoValues(_:_:)` function is identical to the body of the `swapTwoInts(_:_:)` function. However, the first line of `swapTwoValues(_:_:)` is slightly different from `swapTwoInts(_:_:)`. Here's how the first lines compare:

```
1  func swapTwoInts(_ a: inout Int, _ b: inout Int)
2  func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

The generic version of the function uses a *placeholder* type name (called `T`, in this case) instead of an *actual* type name (such as `Int`, `String`, or `Double`). The placeholder type name doesn't say anything about what `T` must be, but it *does* say that both `a` and `b` must be of the same type `T`, whatever `T` represents. The actual type to use in place of `T` is determined each time the `swapTwoValues(_:_:)` function is called.

The other difference between a generic function and a nongeneric function is that the generic function's name (`swapTwoValues(_:_:)`) is followed by the placeholder type name (`T`) inside angle brackets (`<T>`). The brackets tell Swift that `T` is a placeholder type name within the `swapTwoValues(_:_:)` function definition. Because `T` is a placeholder, Swift doesn't look for an actual type called `T`.

The `swapTwoValues(_:_:)` function can now be called in the same way as `swapTwoInts`, except that it can be passed two values of *any* type, as long as both of those values are of the same type as each other. Each time `swapTwoValues(_:_:)` is called, the type to use for `T` is inferred from the types of values passed to the function.

In the two examples below, `T` is inferred to be `Int` and `String` respectively:

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is
      now "hello"
```

> NOTE
>
> The `swapTwoValues(_:_:)` function defined above is inspired by a generic function called `swap`, which is part of the Swift standard library, and is automatically made available for you to use in your apps. If you need the behavior of the `swapTwoValues(_:_:)` function in your own code, you can use Swift's existing `swap(_:_:)` function rather than providing your own implementation.

Type Parameters

In the `swapTwoValues(_:_:)` example above, the placeholder type `T` is an example of a *type parameter*. Type parameters specify and name a placeholder type, and are written immediately after the function's name, between a pair of matching angle brackets (such as `<T>`).

Once you specify a type parameter, you can use it to define the type of a function's parameters (such as the `a` and `b` parameters of the `swapTwoValues(_:_:)` function), or as the function's return type, or as a type annotation within the body of the function. In each case, the type parameter is replaced with an *actual* type whenever the function is called. (In the `swapTwoValues(_:_:)` example above, `T` was replaced with `Int` the first time the function was called, and was replaced with `String` the second time it was called.)

You can provide more than one type parameter by writing multiple type parameter names within the angle brackets, separated by commas.

Naming Type Parameters

In most cases, type parameters have descriptive names, such as `Key` and `Value` in `Dictionary<Key, Value>` and `Element` in `Array<Element>`, which tells the reader about the relationship between the type parameter and the generic type or function it's used in. However, when there isn't a meaningful relationship between them, it's traditional to name them using single letters such as `T`, `U`, and `V`, such as `T` in the `swapTwoValues(_:_:)` function above.

> NOTE
>
> Always give type parameters upper camel case names (such as `T` and `MyTypeParameter`) to indicate that they're a placeholder for a *type*, not a value.

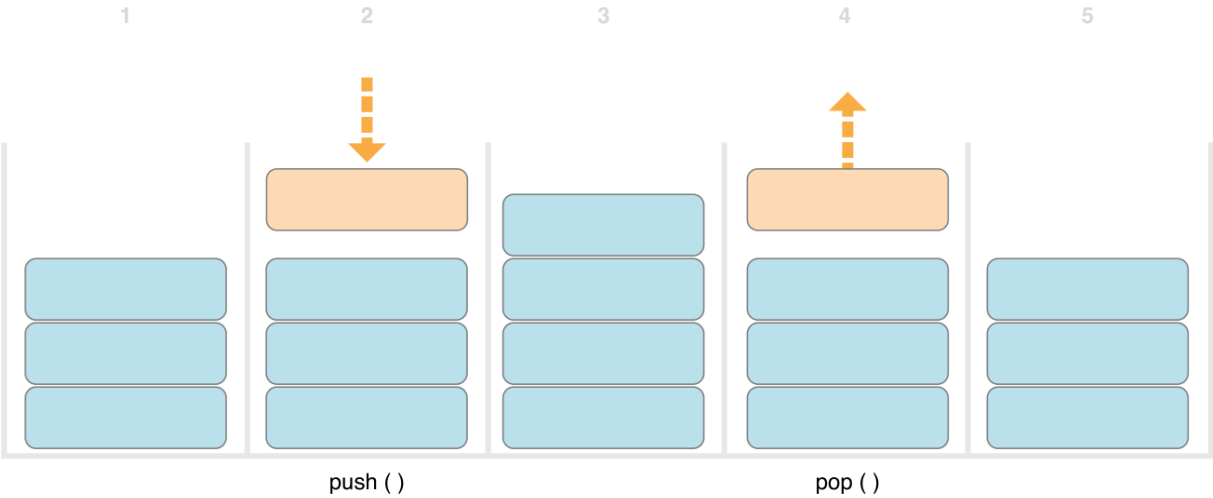
Generic Types

In addition to generic functions, Swift enables you to define your own *generic types*. These are custom classes, structures, and enumerations that can work with *any* type, in a similar way to `Array` and `Dictionary`.

This section shows you how to write a generic collection type called `Stack`. A stack is an ordered set of values, similar to an array, but with a more restricted set of operations than Swift's `Array` type. An array allows new items to be inserted and removed at any location in the array. A stack, however, allows new items to be appended only to the end of the collection (known as *pushing* a new value on to the stack). Similarly, a stack allows items to be removed only from the end of the collection (known as *popping* a value off the stack).

> NOTE
>
> The concept of a stack is used by the `UINavigationController` class to model the view controllers in its navigation hierarchy. You call the `UINavigationController` class `pushViewController(_:animated:)` method to add (or push) a view controller on to the navigation stack, and its `popViewControllerAnimated(_:)` method to remove (or pop) a view controller from the navigation stack. A stack is a useful collection model whenever you need a strict "last in, first out" approach to managing a collection.

The illustration below shows the push and pop behavior for a stack:



1. There are currently three values on the stack.
2. A fourth value is pushed onto the top of the stack.
3. The stack now holds four values, with the most recent one at the top.
4. The top item in the stack is popped.
5. After popping a value, the stack once again holds three values.

Here's how to write a nongeneric version of a stack, in this case for a stack of `Int` values:

```
struct IntStack {
    var items: [Int] = []
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

This structure uses an `Array` property called `items` to store the values in the stack. `Stack` provides two methods, `push` and `pop`, to push and pop values on and off the stack. These methods are marked as `mutating`, because they need to modify (or *mutate*) the structure's `items` array.

The `IntStack` type shown above can only be used with `Int` values, however. It would be much more useful to define a *generic* `Stack` class, that can manage a stack of *any* type of value.

Here's a generic version of the same code:

```
struct Stack<Element> {
    var items: [Element] = []
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}
```

Note how the generic version of `Stack` is essentially the same as the nongeneric version, but with a type parameter called `Element` instead of an actual type of `Int`. This type parameter is written within a pair of angle brackets (`<Element>`) immediately after the structure's name.

`Element` defines a placeholder name for a type to be provided later. This future type can be referred to as `Element` anywhere within the structure's definition. In this case, `Element` is used as a placeholder in three places:

- To create a property called `items`, which is initialized with an empty array of values of type `Element`
- To specify that the `push(_:)` method has a single parameter called `item`, which must be of type `Element`
- To specify that the value returned by the `pop()` method will be a value of type `Element`

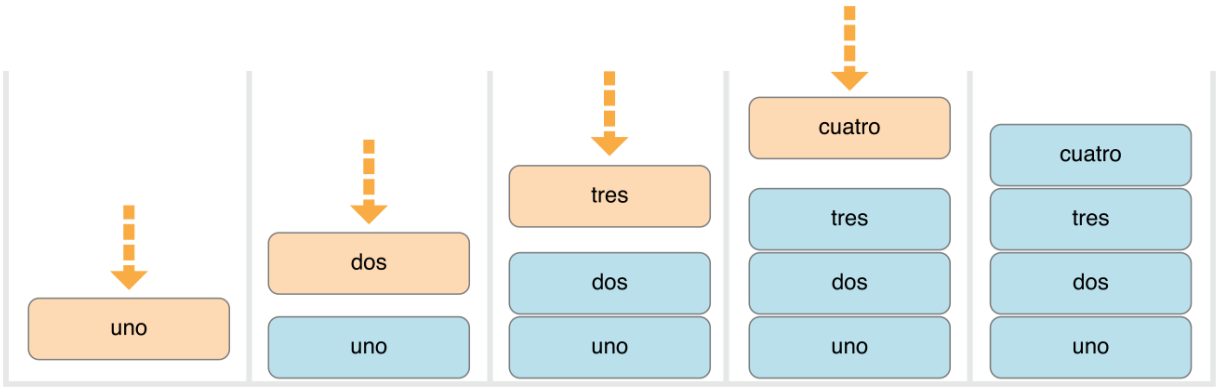
Because it's a generic type, `Stack` can be used to create a stack of *any* valid type in Swift, in a similar manner to `Array` and `Dictionary`.

You create a new `Stack` instance by writing the type to be stored in the stack within angle brackets. For example, to create a new stack of strings, you

write `Stack<String>()`:

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// the stack now contains 4 strings
```

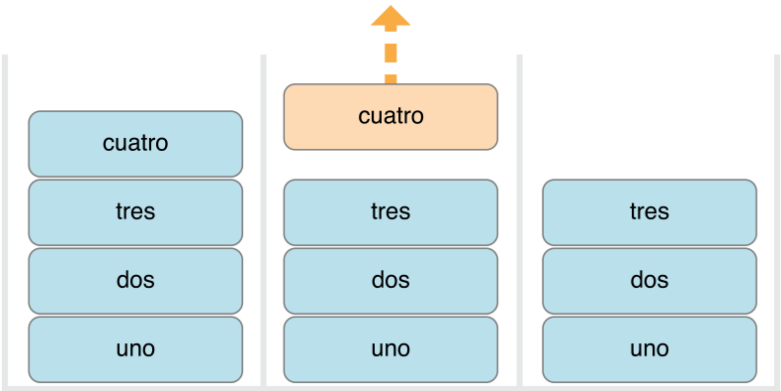
Here's how `stackOfStrings` looks after pushing these four values on to the stack:



Popping a value from the stack removes and returns the top value, `"cuatro"`:

```
let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the stack now
     contains 3 strings
```

Here's how the stack looks after popping its top value:



Extending a Generic Type

When you extend a generic type, you don't provide a type parameter list as part of the extension's definition. Instead, the type parameter list from the *original* type definition is available within the body of the extension, and the original type parameter names are used to refer to the type parameters from the original definition.

The following example extends the generic `Stack` type to add a read-only computed property called `topItem`, which returns the top item on the stack without popping it from the stack:

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil :
     items[items.count - 1]
    }
}
```

The `topItem` property returns an optional value of type `Element`. If the stack is empty, `topItem` returns `nil`; if the stack isn't empty, `topItem` returns the

final item in the `items` array.

Note that this extension doesn't define a type parameter list. Instead, the `Stack` type's existing type parameter name, `Element`, is used within the extension to indicate the optional type of the `topItem` computed property.

The `topItem` computed property can now be used with any `Stack` instance to access and query its top item without removing it.

```
if let topItem = stackOfStrings.topItem {
    print("The top item on the stack is \(topItem).")
}
// Prints "The top item on the stack is tres."
```

Extensions of a generic type can also include requirements that instances of the extended type must satisfy in order to gain the new functionality, as discussed in Extensions with a Generic Where Clause below.

Type Constraints

The `swapTwoValues(_:_:)` function and the `Stack` type can work with any type. However, it's sometimes useful to enforce certain *type constraints* on the types that can be used with generic functions and generic types. Type constraints specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition.

For example, Swift's `Dictionary` type places a limitation on the types that can be used as keys for a dictionary. As described in Dictionaries, the type of a dictionary's keys must be *hashable*. That is, it must provide a way to make itself uniquely representable. `Dictionary` needs its keys to be hashable so that it can check whether it already contains a value for a particular key. Without this requirement, `Dictionary` couldn't tell whether it should insert

or replace a value for a particular key, nor would it be able to find a value for a given key that's already in the dictionary.

This requirement is enforced by a type constraint on the key type for `Dictionary`, which specifies that the key type must conform to the `Hashable` protocol, a special protocol defined in the Swift standard library. All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default. For information about making your own custom types conform to the `Hashable` protocol, see [Conforming to the Hashable Protocol](#).

You can define your own type constraints when creating custom generic types, and these constraints provide much of the power of generic programming. Abstract concepts like `Hashable` characterize types in terms of their conceptual characteristics, rather than their concrete type.

Type Constraint Syntax

You write type constraints by placing a single class or protocol constraint after a type parameter's name, separated by a colon, as part of the type parameter list. The basic syntax for type constraints on a generic function is shown below (although the syntax is the same for generic types):

```
1  func someFunction<T: SomeClass, U: SomeProtocol>
       (someT: T, someU: U) {
2      // function body goes here
3  }
```

The hypothetical function above has two type parameters. The first type parameter, `T`, has a type constraint that requires `T` to be a subclass of `SomeClass`. The second type parameter, `U`, has a type constraint that requires `U` to conform to the protocol `SomeProtocol`.

Type Constraints in Action

Here's a nongeneric function called `findIndex(ofString:in:)`, which is given a `String` value to find and an array of `String` values within which to find it. The `findIndex(ofString:in:)` function returns an optional `Int` value, which will be the index of the first matching string in the array if it's found, or `nil` if the string can't be found:

```
func findIndex(ofString valueToFind: String, in array: [String]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

The `findIndex(ofString:in:)` function can be used to find a string value in an array of strings:

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findIndex(ofString: "llama", in: strings) {
    print("The index of llama is \(foundIndex)")
}
// Prints "The index of llama is 2"
```

The principle of finding the index of a value in an array isn't useful only for strings, however. You can write the same functionality as a generic function

by replacing any mention of strings with values of some type `T` instead.

Here's how you might expect a generic version of `findIndex(ofString:in:)`, called `findIndex(of:in:)`, to be written. Note that the return type of this function is still `Int?`, because the function returns an optional index number, not an optional value from the array. Be warned, though—this function doesn't compile, for reasons explained after the example:

```
1  func findIndex<T>(of valueToFind: T, in array:[T]) ->
       Int? {
2      for (index, value) in array.enumerated() {
3          if value == valueToFind {
4              return index
5          }
6      }
7      return nil
8  }
```

This function doesn't compile as written above. The problem lies with the equality check, "`if value == valueToFind`". Not every type in Swift can be compared with the equal to operator (`==`). If you create your own class or structure to represent a complex data model, for example, then the meaning of "equal to" for that class or structure isn't something that Swift can guess for you. Because of this, it isn't possible to guarantee that this code will work for *every* possible type `T`, and an appropriate error is reported when you try to compile the code.

All is not lost, however. The Swift standard library defines a protocol called `Equatable`, which requires any conforming type to implement the equal to operator (`==`) and the not equal to operator (`!=`) to compare any two values of that type. All of Swift's standard types automatically support the `Equatable` protocol.

Any type that's `Equatable` can be used safely with the `findIndex(of:in:)` function, because it's guaranteed to support the equal to operator. To express this fact, you write a type constraint of `Equatable` as part of the type parameter's definition when you define the function:

```
1  func findIndex<T: Equatable>(of valueToFind: T, in
       array:[T]) -> Int? {
2      for (index, value) in array.enumerated() {
3          if value == valueToFind {
4              return index
5          }
6      }
7      return nil
8  }
```

The single type parameter for `findIndex(of:in:)` is written as `T: Equatable`, which means "any type `T` that conforms to the `Equatable` protocol."

The `findIndex(of:in:)` function now compiles successfully and can be used with any type that's `Equatable`, such as `Double` or `String`:

```
let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])
// doubleIndex is an optional Int with no value, because 9.3 isn't in the array
let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"])
// stringIndex is an optional Int containing a value of 2
```

Associated Types

When defining a protocol, it's sometimes useful to declare one or more associated types as part of the protocol's definition. An *associated type* gives a placeholder name to a type that's used as part of the protocol. The actual type to use for that associated type isn't specified until the protocol is adopted. Associated types are specified with the `associatedtype` keyword.

Associated Types in Action

Here's an example of a protocol called `Container`, which declares an associated type called `Item`:

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

The `Container` protocol defines three required capabilities that any container must provide:

- It must be possible to add a new item to the container with an `append(_:)` method.
- It must be possible to access a count of the items in the container through a `count` property that returns an `Int` value.
- It must be possible to retrieve each item in the container with a subscript that takes an `Int` index value.

This protocol doesn't specify how the items in the container should be stored or what type they're allowed to be. The protocol only specifies the three bits of functionality that any type must provide in order to be considered a `Container`. A conforming type can provide additional functionality, as long as it satisfies these three requirements.

Any type that conforms to the `Container` protocol must be able to specify the type of values it stores. Specifically, it must ensure that only items of the right type are added to the container, and it must be clear about the type of the items returned by its subscript.

To define these requirements, the `Container` protocol needs a way to refer to the type of the elements that a container will hold, without knowing what that type is for a specific container. The `Container` protocol needs to specify that any value passed to the `append(_:)` method must have the same type as

the container's element type, and that the value returned by the container's subscript will be of the same type as the container's element type.

To achieve this, the `Container` protocol declares an associated type called `Item`, written as `associatedtype Item`. The protocol doesn't define what `Item` is—that information is left for any conforming type to provide. Nonetheless, the `Item` alias provides a way to refer to the type of the items in a `Container`, and to define a type for use with the `append(_:)` method and subscript, to ensure that the expected behavior of any `Container` is enforced.

Here's a version of the nongeneric `IntStack` type from Generic Types above, adapted to conform to the `Container` protocol:

```
struct IntStack: Container {
    // original IntStack implementation
    var items: [Int] = []
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias Item = Int
    mutating func append(_ item: Int) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

The `IntStack` type implements all three of the `Container` protocol's requirements, and in each case wraps part of the `IntStack` type's existing functionality to satisfy these requirements.

Moreover, `IntStack` specifies that for this implementation of `Container`, the appropriate `Item` to use is a type of `Int`. The definition of `typealias`

`Item = Int` turns the abstract type of `Item` into a concrete type of `Int` for this implementation of the `Container` protocol.

Thanks to Swift's type inference, you don't actually need to declare a concrete `Item` of `Int` as part of the definition of `IntStack`. Because `IntStack` conforms to all of the requirements of the `Container` protocol, Swift can infer the appropriate `Item` to use, simply by looking at the type of the `append(_:)` method's `item` parameter and the return type of the subscript. Indeed, if you delete the `typealias Item = Int` line from the code above, everything still works, because it's clear what type should be used for `Item`.

You can also make the generic `Stack` type conform to the `Container` protocol:

```swift
struct Stack<Element>: Container {
    // original Stack<Element> implementation
    var items: [Element] = []
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(_ item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }
}
```

This time, the type parameter `Element` is used as the type of the `append(_:)` method's `item` parameter and the return type of the subscript. Swift can therefore infer that `Element` is the appropriate type to use as the `Item` for this particular container.

Extending an Existing Type to Specify an Associated Type

You can extend an existing type to add conformance to a protocol, as described in Adding Protocol Conformance with an Extension. This includes a protocol with an associated type.

Swift's `Array` type already provides an `append(_:)` method, a `count` property, and a subscript with an `Int` index to retrieve its elements. These three capabilities match the requirements of the `Container` protocol. This means that you can extend `Array` to conform to the `Container` protocol simply by declaring that `Array` adopts the protocol. You do this with an empty extension, as described in Declaring Protocol Adoption with an Extension:

```
extension Array: Container {}
```

Array's existing `append(_:)` method and subscript enable Swift to infer the appropriate type to use for `Item`, just as for the generic `Stack` type above. After defining this extension, you can use any `Array` as a `Container`.

Adding Constraints to an Associated Type

You can add type constraints to an associated type in a protocol to require that conforming types satisfy those constraints. For example, the following code defines a version of `Container` that requires the items in the container to be equatable.

```
protocol Container {
    associatedtype Item: Equatable
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

To conform to this version of `Container`, the container's `Item` type has to conform to the `Equatable` protocol.

Using a Protocol in Its Associated Type's Constraints

A protocol can appear as part of its own requirements. For example, here's a protocol that refines the `Container` protocol, adding the requirement of a `suffix(_:)` method. The `suffix(_:)` method returns a given number of elements from the end of the container, storing them in an instance of the `Suffix` type.

```
protocol SuffixableContainer: Container {
    associatedtype Suffix: SuffixableContainer where
     Suffix.Item == Item
    func suffix(_ size: Int) -> Suffix
}
```

In this protocol, `Suffix` is an associated type, like the `Item` type in the `Container` example above. `Suffix` has two constraints: It must conform to the `SuffixableContainer` protocol (the protocol currently being defined), and its `Item` type must be the same as the container's `Item` type. The constraint on `Item` is a generic `where` clause, which is discussed in Associated Types with a Generic Where Clause below.

Here's an extension of the `Stack` type from Generic Types above that adds conformance to the `SuffixableContainer` protocol:

```
extension Stack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack {
        var result = Stack()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack.
}
var stackOfInts = Stack<Int>()
stackOfInts.append(10)
stackOfInts.append(20)
stackOfInts.append(30)
let suffix = stackOfInts.suffix(2)
// suffix contains 20 and 30
```

In the example above, the `Suffix` associated type for `Stack` is also `Stack`, so the suffix operation on `Stack` returns another `Stack`. Alternatively, a type that conforms to `SuffixableContainer` can have a `Suffix` type that's different from itself—meaning the suffix operation can return a different type. For example, here's an extension to the nongeneric `IntStack` type that adds `SuffixableContainer` conformance, using `Stack<Int>` as its suffix type instead of `IntStack`:

```
extension IntStack: SuffixableContainer {
    func suffix(_ size: Int) -> Stack<Int> {
        var result = Stack<Int>()
        for index in (count-size)..<count {
            result.append(self[index])
        }
        return result
    }
    // Inferred that Suffix is Stack<Int>.
}
```

Generic Where Clauses

Type constraints, as described in Type Constraints, enable you to define requirements on the type parameters associated with a generic function, subscript, or type.

It can also be useful to define requirements for associated types. You do this by defining a *generic where clause*. A generic `where` clause enables you to require that an associated type must conform to a certain protocol, or that certain type parameters and associated types must be the same. A generic `where` clause starts with the `where` keyword, followed by constraints for associated types or equality relationships between types and associated types. You write a generic `where` clause right before the opening curly brace of a type or function's body.

The example below defines a generic function called `allItemsMatch`, which checks to see if two `Container` instances contain the same items in the same order. The function returns a Boolean value of `true` if all items match and a value of `false` if they don't.

The two containers to be checked don't have to be the same type of container (although they can be), but they do have to hold the same type of items. This requirement is expressed through a combination of type constraints and a generic `where` clause:

```swift
func allItemsMatch<C1: Container, C2: Container>
    (_ someContainer: C1, _ anotherContainer: C2) -> Bool
    where C1.Item == C2.Item, C1.Item: Equatable {

        // Check that both containers contain the same number of items.
        if someContainer.count != anotherContainer.count {
            return false
        }

        // Check each pair of items to see if they're equivalent.
        for i in 0..<someContainer.count {
            if someContainer[i] != anotherContainer[i] {
                return false
            }
        }

        // All items match, so return true.
        return true
}
```

This function takes two arguments called `someContainer` and `anotherContainer`. The `someContainer` argument is of type `C1`, and the

`anotherContainer` argument is of type `C2`. Both `C1` and `C2` are type parameters for two container types to be determined when the function is called.

The following requirements are placed on the function's two type parameters:

- `C1` must conform to the `Container` protocol (written as `C1: Container`).
- `C2` must also conform to the `Container` protocol (written as `C2: Container`).
- The `Item` for `C1` must be the same as the `Item` for `C2` (written as `C1.Item == C2.Item`).
- The `Item` for `C1` must conform to the `Equatable` protocol (written as `C1.Item: Equatable`).

The first and second requirements are defined in the function's type parameter list, and the third and fourth requirements are defined in the function's generic `where` clause.

These requirements mean:

- `someContainer` is a container of type `C1`.
- `anotherContainer` is a container of type `C2`.
- `someContainer` and `anotherContainer` contain the same type of items.
- The items in `someContainer` can be checked with the not equal operator (`!=`) to see if they're different from each other.

The third and fourth requirements combine to mean that the items in `anotherContainer` can *also* be checked with the `!=` operator, because they're exactly the same type as the items in `someContainer`.

These requirements enable the `allItemsMatch(_:_:)` function to compare the two containers, even if they're of a different container type.

The `allItemsMatch(_:_:)` function starts by checking that both containers contain the same number of items. If they contain a different number of items, there's no way that they can match, and the function returns `false`.

After making this check, the function iterates over all of the items in `someContainer` with a `for-in` loop and the half-open range operator (`..<`). For each item, the function checks whether the item from `someContainer` isn't equal to the corresponding item in `anotherContainer`. If the two items aren't equal, then the two containers don't match, and the function returns `false`.

If the loop finishes without finding a mismatch, the two containers match, and the function returns `true`.

Here's how the `allItemsMatch(_:_:)` function looks in action:

```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    print("All items match.")
} else {
    print("Not all items match.")
}
// Prints "All items match."
```

The example above creates a `Stack` instance to store `String` values, and pushes three strings onto the stack. The example also creates an `Array` instance initialized with an array literal containing the same three strings as the stack. Even though the stack and the array are of a different type, they both conform to the `Container` protocol, and both contain the same type of values. You can therefore call the `allItemsMatch(_:_:)` function with these two containers as its arguments. In the example above, the `allItemsMatch(_:_:)` function correctly reports that all of the items in the two containers match.

Extensions with a Generic Where Clause

You can also use a generic `where` clause as part of an extension. The example below extends the generic `Stack` structure from the previous examples to add an `isTop(_:)` method.

```
extension Stack where Element: Equatable {
    func isTop(_ item: Element) -> Bool {
        guard let topItem = items.last else {
            return false
        }
        return topItem == item
    }
}
```

This new `isTop(_:)` method first checks that the stack isn't empty, and then compares the given item against the stack's topmost item. If you tried to do this without a generic `where` clause, you would have a problem: The implementation of `isTop(_:)` uses the `==` operator, but the definition of `Stack` doesn't require its items to be equatable, so using the `==` operator

results in a compile-time error. Using a generic `where` clause lets you add a new requirement to the extension, so that the extension adds the `isTop(_:)` method only when the items in the stack are equatable.

Here's how the `isTop(_:)` method looks in action:

```
if stackOfStrings.isTop("tres") {
    print("Top element is tres.")
} else {
    print("Top element is something else.")
}
// Prints "Top element is tres."
```

If you try to call the `isTop(_:)` method on a stack whose elements aren't equatable, you'll get a compile-time error.

```
struct NotEquatable { }
var notEquatableStack = Stack<NotEquatable>()
let notEquatableValue = NotEquatable()
notEquatableStack.push(notEquatableValue)
notEquatableStack.isTop(notEquatableValue)  // Error
```

You can use a generic `where` clause with extensions to a protocol. The example below extends the `Container` protocol from the previous examples to add a `startsWith(_:)` method.

```
extension Container where Item: Equatable {
    func startsWith(_ item: Item) -> Bool {
        return count >= 1 && self[0] == item
    }
}
```

The `startsWith(_:)` method first makes sure that the container has at least one item, and then it checks whether the first item in the container matches the given item. This new `startsWith(_:)` method can be used with any type that conforms to the `Container` protocol, including the stacks and arrays used above, as long as the container's items are equatable.

```
if [9, 9, 9].startsWith(42) {
    print("Starts with 42.")
} else {
    print("Starts with something else.")
}
// Prints "Starts with something else."
```

The generic `where` clause in the example above requires `Item` to conform to a protocol, but you can also write a generic `where` clauses that require `Item` to be a specific type. For example:

```swift
extension Container where Item == Double {
    func average() -> Double {
        var sum = 0.0
        for index in 0..<count {
            sum += self[index]
        }
        return sum / Double(count)
    }
}
print([1260.0, 1200.0, 98.6, 37.0].average())
// Prints "648.9"
```

This example adds an `average()` method to containers whose `Item` type is `Double`. It iterates over the items in the container to add them up, and divides by the container's count to compute the average. It explicitly converts the count from `Int` to `Double` to be able to do floating-point division.

You can include multiple requirements in a generic `where` clause that's part of an extension, just like you can for a generic `where` clause that you write elsewhere. Separate each requirement in the list with a comma.

Contextual Where Clauses

You can write a generic `where` clause as part of a declaration that doesn't have its own generic type constraints, when you're already working in the context of generic types. For example, you can write a generic `where` clause on a subscript of a generic type or on a method in an extension to a generic type. The `Container` structure is generic, and the `where` clauses in the

example below specify what type constraints have to be satisfied to make these new methods available on a container.

```
extension Container {
    func average() -> Double where Item == Int {
        var sum = 0.0
        for index in 0..<count {
            sum += Double(self[index])
        }
        return sum / Double(count)
    }
    func endsWith(_ item: Item) -> Bool where Item: Equatable {
        return count >= 1 && self[count-1] == item
    }
}
let numbers = [1260, 1200, 98, 37]
print(numbers.average())
// Prints "648.75"
print(numbers.endsWith(37))
// Prints "true"
```

This example adds an `average()` method to `Container` when the items are integers, and it adds an `endsWith(_:)` method when the items are equatable. Both functions include a generic `where` clause that adds type constraints to the generic `Item` type parameter from the original declaration of `Container`.

If you want to write this code without using contextual `where` clauses, you write two extensions, one for each generic `where` clause. The example above

and the example below have the same behavior.

```
extension Container where Item == Int {
    func average() -> Double {
        var sum = 0.0
        for index in 0..<count {
            sum += Double(self[index])
        }
        return sum / Double(count)
    }
}
extension Container where Item: Equatable {
    func endsWith(_ item: Item) -> Bool {
        return count >= 1 && self[count-1] == item
    }
}
```

In the version of this example that uses contextual `where` clauses, the implementation of `average()` and `endsWith(_:)` are both in the same extension because each method's generic `where` clause states the requirements that need to be satisfied to make that method available. Moving those requirements to the extensions' generic `where` clauses makes the methods available in the same situations, but requires one extension per requirement.

Associated Types with a Generic Where Clause

You can include a generic `where` clause on an associated type. For example, suppose you want to make a version of `Container` that includes an iterator, like what the `Sequence` protocol uses in the standard library. Here's how you write that:

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }

    associatedtype Iterator: IteratorProtocol where
     Iterator.Element == Item
    func makeIterator() -> Iterator
}
```

The generic `where` clause on `Iterator` requires that the iterator must traverse over elements of the same item type as the container's items, regardless of the iterator's type. The `makeIterator()` function provides access to a container's iterator.

For a protocol that inherits from another protocol, you add a constraint to an inherited associated type by including the generic `where` clause in the protocol declaration. For example, the following code declares a `ComparableContainer` protocol that requires `Item` to conform to `Comparable`:

```
protocol ComparableContainer: Container where Item:
     Comparable { }
```

Generic Subscripts

Subscripts can be generic, and they can include generic `where` clauses. You write the placeholder type name inside angle brackets after `subscript`, and you write a generic `where` clause right before the opening curly brace of the subscript's body. For example:

```
extension Container {
    subscript<Indices: Sequence>(indices: Indices) -> [Item]
        where Indices.Iterator.Element == Int {
            var result: [Item] = []
            for index in indices {
                result.append(self[index])
            }
            return result
    }
}
```

This extension to the `Container` protocol adds a subscript that takes a sequence of indices and returns an array containing the items at each given index. This generic subscript is constrained as follows:

- The generic parameter `Indices` in angle brackets has to be a type that conforms to the `Sequence` protocol from the standard library.
- The subscript takes a single parameter, `indices`, which is an instance of that `Indices` type.
- The generic `where` clause requires that the iterator for the sequence must traverse over elements of type `Int`. This ensures that the indices in the sequence are the same type as the indices used for a container.

Taken together, these constraints mean that the value passed for the `indices` parameter is a sequence of integers.

Opaque Types

A function or method with an opaque return type hides its return value's type information. Instead of providing a concrete type as the function's return type, the return value is described in terms of the protocols it supports. Hiding type information is useful at boundaries between a module and code that calls into the module, because the underlying type of the return value can remain private. Unlike returning a value whose type is a protocol type, opaque types preserve type identity—the compiler has access to the type information, but clients of the module don't.

The Problem That Opaque Types Solve

For example, suppose you're writing a module that draws ASCII art shapes. The basic characteristic of an ASCII art shape is a `draw()` function that returns the string representation of that shape, which you can use as the requirement for the `Shape` protocol:

```
protocol Shape {
    func draw() -> String
}

struct Triangle: Shape {
    var size: Int
    func draw() -> String {
        var result: [String] = []
        for length in 1...size {
            result.append(String(repeating: "*", count: length))
        }
        return result.joined(separator: "\n")
    }
}
let smallTriangle = Triangle(size: 3)
print(smallTriangle.draw())
// *
// **
// ***
```

You could use generics to implement operations like flipping a shape vertically, as shown in the code below. However, there's an important limitation to this approach: The flipped result exposes the exact generic types that were used to create it.

```
struct FlippedShape<T: Shape>: Shape {
    var shape: T
    func draw() -> String {
        let lines = shape.draw().split(separator: "\n")
        return lines.reversed().joined(separator: "\n")
    }
}
let flippedTriangle = FlippedShape(shape: smallTriangle)
print(flippedTriangle.draw())
// ***
// **
// *
```

This approach to defining a `JoinedShape<T: Shape, U: Shape>` structure that joins two shapes together vertically, like the code below shows, results in types like `JoinedShape<FlippedShape<Triangle>, Triangle>` from joining a flipped triangle with another triangle.

```swift
struct JoinedShape<T: Shape, U: Shape>: Shape {
    var top: T
    var bottom: U
    func draw() -> String {
        return top.draw() + "\n" + bottom.draw()
    }
}
let joinedTriangles = JoinedShape(top:
     smallTriangle, bottom: flippedTriangle)
print(joinedTriangles.draw())
// *
// **
// ***
// ***
// **
// *
```

Exposing detailed information about the creation of a shape allows types that aren't meant to be part of the ASCII art module's public interface to leak out because of the need to state the full return type. The code inside the module could build up the same shape in a variety of ways, and other code outside the module that uses the shape shouldn't have to account for the implementation details about the list of transformations. Wrapper types like `JoinedShape` and `FlippedShape` don't matter to the module's users, and they shouldn't be visible. The module's public interface consists of operations like joining and flipping a shape, and those operations return another `Shape` value.

Returning an Opaque Type

You can think of an opaque type like being the reverse of a generic type. Generic types let the code that calls a function pick the type for that function's parameters and return value in a way that's abstracted away from the function implementation. For example, the function in the following code returns a type that depends on its caller:

```
func max<T>(_ x: T, _ y: T) -> T where T: Comparable
    { ... }
```

The code that calls `max(_:_:)` chooses the values for `x` and `y`, and the type of those values determines the concrete type of `T`. The calling code can use any type that conforms to the `Comparable` protocol. The code inside the function is written in a general way so it can handle whatever type the caller provides. The implementation of `max(_:_:)` uses only functionality that all `Comparable` types share.

Those roles are reversed for a function with an opaque return type. An opaque type lets the function implementation pick the type for the value it returns in a way that's abstracted away from the code that calls the function. For example, the function in the following example returns a trapezoid without exposing the underlying type of that shape.

```
struct Square: Shape {
    var size: Int
    func draw() -> String {
        let line = String(repeating: "*", count: size)
        let result = Array<String>(repeating: line, count: size)
        return result.joined(separator: "\n")
    }
}

func makeTrapezoid() -> some Shape {
    let top = Triangle(size: 2)
    let middle = Square(size: 2)
    let bottom = FlippedShape(shape: top)
    let trapezoid = JoinedShape(
        top: top,
        bottom: JoinedShape(top: middle, bottom: bottom)
    )
    return trapezoid
}
let trapezoid = makeTrapezoid()
print(trapezoid.draw())
// *
// **
```

```
// **
// **
// **
// *
```

The `makeTrapezoid()` function in this example declares its return type as `some Shape`; as a result, the function returns a value of some given type that conforms to the `Shape` protocol, without specifying any particular concrete type. Writing `makeTrapezoid()` this way lets it express the fundamental aspect of its public interface—the value it returns is a shape—without making the specific types that the shape is made from a part of its public interface. This implementation uses two triangles and a square, but the function could be rewritten to draw a trapezoid in a variety of other ways without changing its return type.

This example highlights the way that an opaque return type is like the reverse of a generic type. The code inside `makeTrapezoid()` can return any type it needs to, as long as that type conforms to the `Shape` protocol, like the calling code does for a generic function. The code that calls the function needs to be written in a general way, like the implementation of a generic function, so that it can work with any `Shape` value that's returned by `makeTrapezoid()`.

You can also combine opaque return types with generics. The functions in the following code both return a value of some type that conforms to the `Shape` protocol.

```swift
func flip<T: Shape>(_ shape: T) -> some Shape {
    return FlippedShape(shape: shape)
}
func join<T: Shape, U: Shape>(_ top: T, _ bottom: U)
     -> some Shape {
    JoinedShape(top: top, bottom: bottom)
}

let opaqueJoinedTriangles = join(smallTriangle,
     flip(smallTriangle))
print(opaqueJoinedTriangles.draw())
// *
// **
// ***
// ***
// **
// *
```

The value of `opaqueJoinedTriangles` in this example is the same as `joinedTriangles` in the generics example in the [The Problem That Opaque Types Solve]() section earlier in this chapter. However, unlike the value in that example, `flip(_:)` and `join(_:_:)` wrap the underlying types that the generic shape operations return in an opaque return type, which prevents those types from being visible. Both functions are generic because the types they rely on are generic, and the type parameters to the function pass along the type information needed by `FlippedShape` and `JoinedShape`.

If a function with an opaque return type returns from multiple places, all of the possible return values must have the same type. For a generic function, that return type can use the function's generic type parameters, but it must still be a single type. For example, here's an *invalid* version of the shape-flipping function that includes a special case for squares:

```
1  func invalidFlip<T: Shape>(_ shape: T) -> some Shape
       {
2      if shape is Square {
3          return shape // Error: return types don't
       match
4      }
5      return FlippedShape(shape: shape) // Error:
       return types don't match
6  }
```

If you call this function with a `Square`, it returns a `Square`; otherwise, it returns a `FlippedShape`. This violates the requirement to return values of only one type and makes `invalidFlip(_:)` invalid code. One way to fix `invalidFlip(_:)` is to move the special case for squares into the implementation of `FlippedShape`, which lets this function always return a `FlippedShape` value:

```
struct FlippedShape<T: Shape>: Shape {
    var shape: T
    func draw() -> String {
        if shape is Square {
            return shape.draw()
        }
        let lines = shape.draw().split(separator: "\n")
        return lines.reversed().joined(separator: "\n")
    }
}
```

The requirement to always return a single type doesn't prevent you from using generics in an opaque return type. Here's an example of a function that incorporates its type parameter into the underlying type of the value it returns:

```
func `repeat`<T: Shape>(shape: T, count: Int) -> some Collection {
    return Array<T>(repeating: shape, count: count)
}
```

In this case, the underlying type of the return value varies depending on `T`: Whatever shape is passed it, `repeat(shape:count:)` creates and returns an array of that shape. Nevertheless, the return value always has the same underlying type of `[T]`, so it follows the requirement that functions with opaque return types must return values of only a single type.

Differences Between Opaque Types and Protocol Types

Returning an opaque type looks very similar to using a protocol type as the return type of a function, but these two kinds of return type differ in whether they preserve type identity. An opaque type refers to one specific type, although the caller of the function isn't able to see which type; a protocol type can refer to any type that conforms to the protocol. Generally speaking, protocol types give you more flexibility about the underlying types of the values they store, and opaque types let you make stronger guarantees about those underlying types.

For example, here's a version of `flip(_:)` that uses a protocol type as its return type instead of an opaque return type:

```
func protoFlip<T: Shape>(_ shape: T) -> Shape {
    return FlippedShape(shape: shape)
}
```

This version of `protoFlip(_:)` has the same body as `flip(_:)`, and it always returns a value of the same type. Unlike `flip(_:)`, the value that `protoFlip(_:)` returns isn't required to always have the same type—it just has to conform to the `Shape` protocol. Put another way, `protoFlip(_:)` makes a much looser API contract with its caller than `flip(_:)` makes. It reserves the flexibility to return values of multiple types:

```
func protoFlip<T: Shape>(_ shape: T) -> Shape {
    if shape is Square {
        return shape
    }

    return FlippedShape(shape: shape)
}
```

The revised version of the code returns an instance of `Square` or an instance of `FlippedShape`, depending on what shape is passed in. Two flipped shapes returned by this function might have completely different types. Other valid versions of this function could return values of different types when flipping multiple instances of the same shape. The less specific return type information from `protoFlip(_:)` means that many operations that depend on type information aren't available on the returned value. For example, it's not possible to write an `==` operator comparing results returned by this function.

```
let protoFlippedTriangle = protoFlip(smallTriangle)
let sameThing = protoFlip(smallTriangle)
protoFlippedTriangle == sameThing  // Error
```

The error on the last line of the example occurs for several reasons. The immediate issue is that the `Shape` doesn't include an `==` operator as part of its protocol requirements. If you try adding one, the next issue you'll encounter is that the `==` operator needs to know the types of its left-hand and right-hand arguments. This sort of operator usually takes arguments of type `Self`, matching whatever concrete type adopts the protocol, but adding a `Self` requirement to the protocol doesn't allow for the type erasure that happens when you use the protocol as a type.

Using a protocol type as the return type for a function gives you the flexibility to return any type that conforms to the protocol. However, the cost of that flexibility is that some operations aren't possible on the returned values. The example shows how the `==` operator isn't available—it depends on specific type information that isn't preserved by using a protocol type.

Another problem with this approach is that the shape transformations don't nest. The result of flipping a triangle is a value of type `Shape`, and the `protoFlip(_:)` function takes an argument of some type that conforms to the `Shape` protocol. However, a value of a protocol type doesn't conform to that protocol; the value returned by `protoFlip(_:)` doesn't conform to `Shape`. This means code like `protoFlip(protoFlip(smallTriange))` that

applies multiple transformations is invalid because the flipped shape isn't a valid argument to `protoFlip(_:)`.

In contrast, opaque types preserve the identity of the underlying type. Swift can infer associated types, which lets you use an opaque return value in places where a protocol type can't be used as a return value. For example, here's a version of the `Container` protocol from Generics:

```
protocol Container {
    associatedtype Item
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
extension Array: Container { }
```

You can't use `Container` as the return type of a function because that protocol has an associated type. You also can't use it as constraint in a generic return type because there isn't enough information outside the function body to infer what the generic type needs to be.

```
// Error: Protocol with associated types can't be used as a return type.
func makeProtocolContainer<T>(item: T) -> Container {
    return [item]
}

// Error: Not enough information to infer C.
func makeProtocolContainer<T, C: Container>(item: T) -> C {
    return [item]
}
```

Using the opaque type `some Container` as a return type expresses the desired API contract—the function returns a container, but declines to specify the container's type:

```
func makeOpaqueContainer<T>(item: T) -> some Container {
    return [item]
}
let opaqueContainer = makeOpaqueContainer(item: 12)
let twelve = opaqueContainer[0]
print(type(of: twelve))
// Prints "Int"
```

The type of `twelve` is inferred to be `Int`, which illustrates the fact that type inference works with opaque types. In the implementation of

`makeOpaqueContainer(item:)`, the underlying type of the opaque container is `[T]`. In this case, `T` is `Int`, so the return value is an array of integers and the `Item` associated type is inferred to be `Int`. The subscript on `Container` returns `Item`, which means that the type of `twelve` is also inferred to be `Int`.

Automatic Reference Counting

Swift uses *Automatic Reference Counting* (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you don't need to think about memory management yourself. ARC automatically frees up the memory used by class instances when those instances are no longer needed.

However, in a few cases ARC requires more information about the relationships between parts of your code in order to manage memory for you. This chapter describes those situations and shows how you enable ARC to manage all of your app's memory. Using ARC in Swift is very similar to the approach described in Transitioning to ARC Release Notes for using ARC with Objective-C.

Reference counting applies only to instances of classes. Structures and enumerations are value types, not reference types, and aren't stored and passed by reference.

How ARC Works

Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances don't take up space in memory when they're no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance's properties, or call that instance's methods. Indeed, if you tried to access the instance, your app would most likely crash.

To make sure that instances don't disappear while they're still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists.

To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a *strong reference* to the instance. The reference is called a "strong" reference because it keeps a firm hold on that instance, and doesn't allow it to be deallocated for as long as that strong reference remains.

ARC in Action

Here's an example of how Automatic Reference Counting works. This example starts with a simple class called `Person`, which defines a stored constant property called `name`:

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
```

The `Person` class has an initializer that sets the instance's `name` property and prints a message to indicate that initialization is underway. The `Person` class also has a deinitializer that prints a message when an instance of the class is deallocated.

The next code snippet defines three variables of type `Person?`, which are used to set up multiple references to a new `Person` instance in subsequent code snippets. Because these variables are of an optional type (`Person?`, not `Person`), they're automatically initialized with a value of `nil`, and don't currently reference a `Person` instance.

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

You can now create a new `Person` instance and assign it to one of these three variables:

```
reference1 = Person(name: "John Appleseed")
// Prints "John Appleseed is being initialized"
```

Note that the message `"John Appleseed is being initialized"` is printed at the point that you call the `Person` class's initializer. This confirms that initialization has taken place.

Because the new `Person` instance has been assigned to the `reference1` variable, there's now a strong reference from `reference1` to the new `Person` instance. Because there's at least one strong reference, ARC makes sure that this `Person` is kept in memory and isn't deallocated.

If you assign the same `Person` instance to two more variables, two more strong references to that instance are established:

```
reference2 = reference1
reference3 = reference1
```

There are now *three* strong references to this single `Person` instance.

If you break two of these strong references (including the original reference) by assigning `nil` to two of the variables, a single strong reference remains, and the `Person` instance isn't deallocated:

```
reference1 = nil
reference2 = nil
```

ARC doesn't deallocate the `Person` instance until the third and final strong reference is broken, at which point it's clear that you are no longer using the `Person` instance:

```
reference3 = nil
// Prints "John Appleseed is being deinitialized"
```

Strong Reference Cycles Between Class Instances

In the examples above, ARC is able to track the number of references to the new `Person` instance you create and to deallocate that `Person` instance when it's no longer needed.

However, it's possible to write code in which an instance of a class *never* gets to a point where it has zero strong references. This can happen if two class instances hold a strong reference to each other, such that each instance keeps the other alive. This is known as a *strong reference cycle*.

You resolve strong reference cycles by defining some of the relationships between classes as weak or unowned references instead of as strong references. This process is described in Resolving Strong Reference Cycles Between Class Instances. However, before you learn how to resolve a strong reference cycle, it's useful to understand how such a cycle is caused.

Here's an example of how a strong reference cycle can be created by accident. This example defines two classes called `Person` and `Apartment`, which model a block of apartments and its residents:

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized")
     }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being
     deinitialized") }
}
```

Every `Person` instance has a `name` property of type `String` and an optional `apartment` property that's initially `nil`. The `apartment` property is optional, because a person may not always have an apartment.

Similarly, every `Apartment` instance has a `unit` property of type `String` and has an optional `tenant` property that's initially `nil`. The tenant property is optional because an apartment may not always have a tenant.

Both of these classes also define a deinitializer, which prints the fact that an instance of that class is being deinitialized. This enables you to see whether instances of `Person` and `Apartment` are being deallocated as expected.

This next code snippet defines two variables of optional type called `john` and `unit4A`, which will be set to a specific `Apartment` and `Person` instance

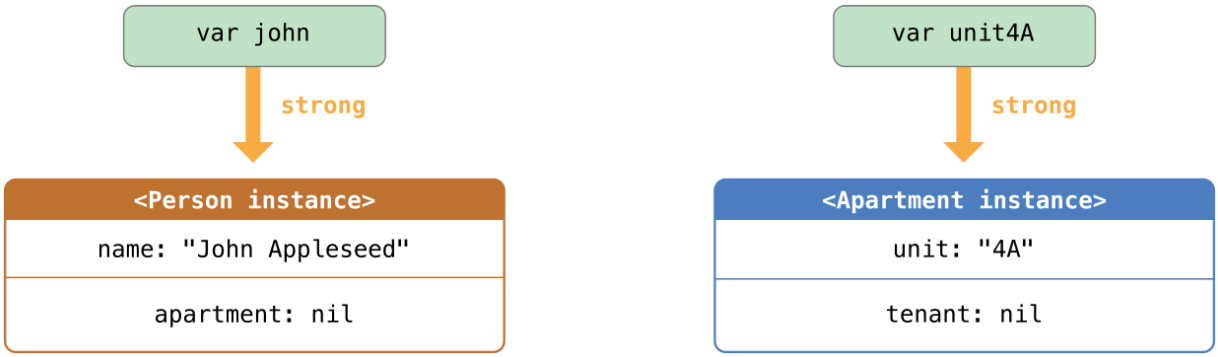
below. Both of these variables have an initial value of `nil`, by virtue of being optional:

```
var john: Person?
var unit4A: Apartment?
```

You can now create a specific `Person` instance and `Apartment` instance and assign these new instances to the `john` and `unit4A` variables:

```
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
```

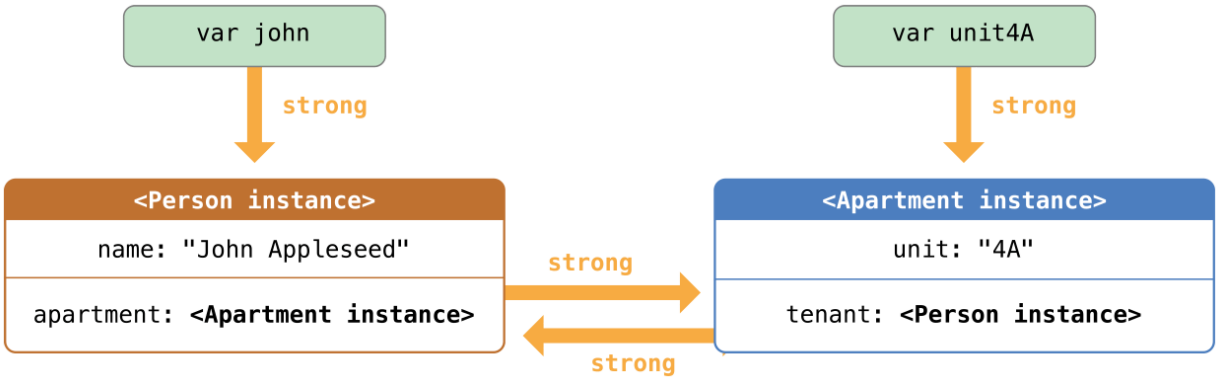
Here's how the strong references look after creating and assigning these two instances. The `john` variable now has a strong reference to the new `Person` instance, and the `unit4A` variable has a strong reference to the new `Apartment` instance:



You can now link the two instances together so that the person has an apartment, and the apartment has a tenant. Note that an exclamation point (`!`) is used to unwrap and access the instances stored inside the `john` and `unit4A` optional variables, so that the properties of those instances can be set:

```
john!.apartment = unit4A
unit4A!.tenant = john
```

Here's how the strong references look after you link the two instances together:

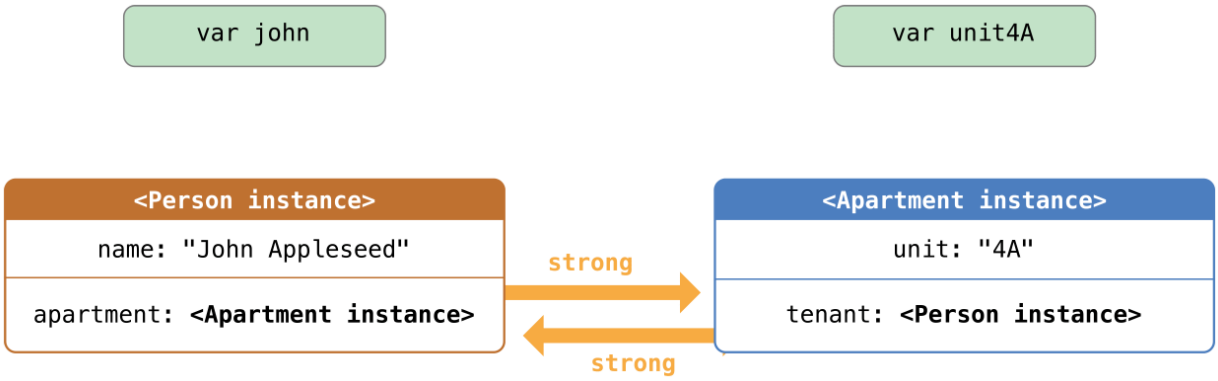


Unfortunately, linking these two instances creates a strong reference cycle between them. The `Person` instance now has a strong reference to the `Apartment` instance, and the `Apartment` instance has a strong reference to the `Person` instance. Therefore, when you break the strong references held by the `john` and `unit4A` variables, the reference counts don't drop to zero, and the instances aren't deallocated by ARC:

```
john = nil
unit4A = nil
```

Note that neither deinitializer was called when you set these two variables to `nil`. The strong reference cycle prevents the `Person` and `Apartment` instances from ever being deallocated, causing a memory leak in your app.

Here's how the strong references look after you set the `john` and `unit4A` variables to `nil`:



The strong references between the `Person` instance and the `Apartment` instance remain and can't be broken.

Resolving Strong Reference Cycles Between Class Instances

Swift provides two ways to resolve strong reference cycles when you work with properties of class type: weak references and unowned references.

Weak and unowned references enable one instance in a reference cycle to refer to the other instance *without* keeping a strong hold on it. The instances can then refer to each other without creating a strong reference cycle.

Use a weak reference when the other instance has a shorter lifetime—that is, when the other instance can be deallocated first. In the `Apartment` example above, it's appropriate for an apartment to be able to have no tenant at some point in its lifetime, and so a weak reference is an appropriate way to break the reference cycle in this case. In contrast, use an unowned reference when the other instance has the same lifetime or a longer lifetime.

Weak References

A *weak reference* is a reference that doesn't keep a strong hold on the instance it refers to, and so doesn't stop ARC from disposing of the referenced instance. This behavior prevents the reference from becoming part of a strong reference cycle. You indicate a weak reference by placing the `weak` keyword before a property or variable declaration.

Because a weak reference doesn't keep a strong hold on the instance it refers to, it's possible for that instance to be deallocated while the weak reference is still referring to it. Therefore, ARC automatically sets a weak reference to `nil` when the instance that it refers to is deallocated. And, because weak references need to allow their value to be changed to `nil` at runtime, they're always declared as variables, rather than constants, of an optional type.

You can check for the existence of a value in the weak reference, just like any other optional value, and you will never end up with a reference to an invalid instance that no longer exists.

> NOTE
>
> Property observers aren't called when ARC sets a weak reference to `nil`.

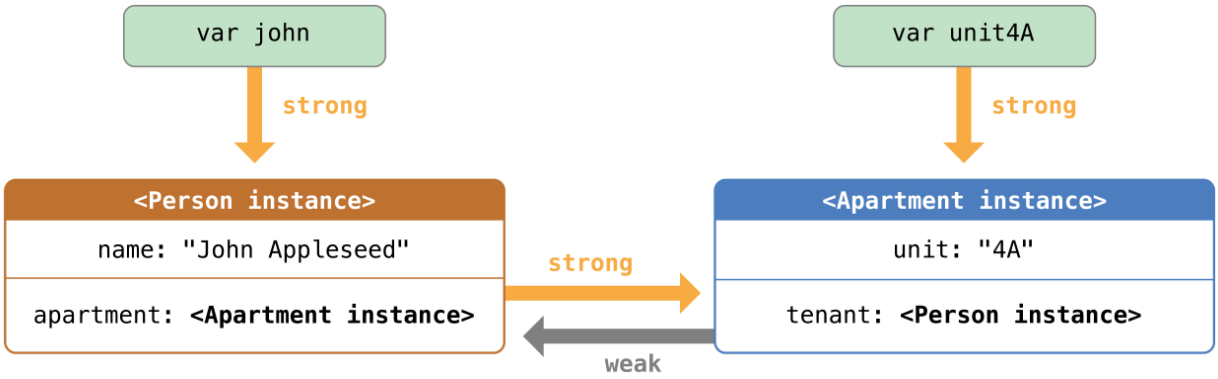
The example below is identical to the `Person` and `Apartment` example from above, with one important difference. This time around, the `Apartment` type's `tenant` property is declared as a weak reference:

```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```

The strong references from the two variables (`john` and `unit4A`) and the links between the two instances are created as before:

```
var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john
```

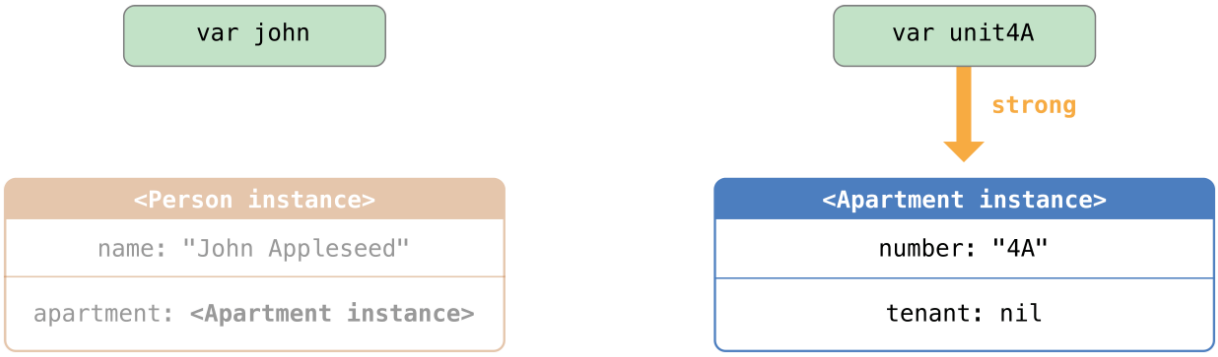
Here's how the references look now that you've linked the two instances together:



The `Person` instance still has a strong reference to the `Apartment` instance, but the `Apartment` instance now has a *weak* reference to the `Person` instance. This means that when you break the strong reference held by the `john` variable by setting it to `nil`, there are no more strong references to the `Person` instance:

```
1  john = nil
2  // Prints "John Appleseed is being deinitialized"
```

Because there are no more strong references to the `Person` instance, it's deallocated and the `tenant` property is set to `nil`:

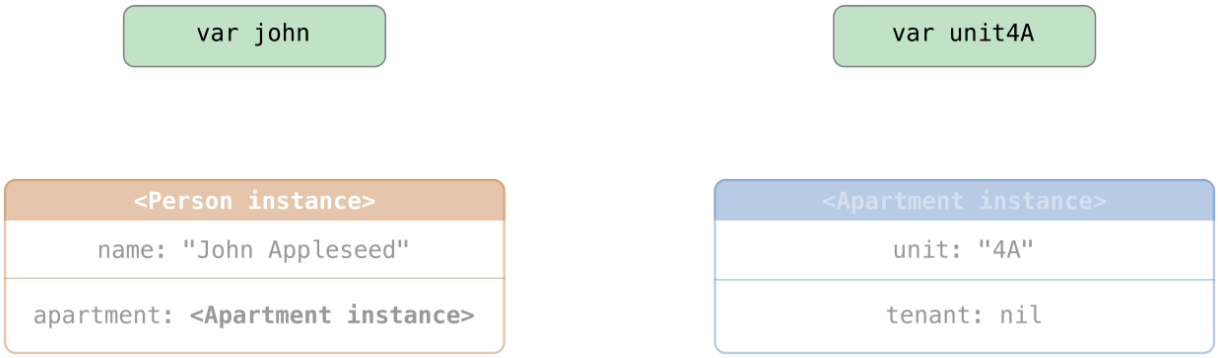


The only remaining strong reference to the `Apartment` instance is from the `unit4A` variable. If you break *that* strong reference, there are no more strong

references to the `Apartment` instance:

```
unit4A = nil
// Prints "Apartment 4A is being deinitialized"
```

Because there are no more strong references to the `Apartment` instance, it too is deallocated:



> NOTE
>
> In systems that use garbage collection, weak pointers are sometimes used to implement a simple caching mechanism because objects with no strong references are deallocated only when memory pressure triggers garbage collection. However, with ARC, values are deallocated as soon as their last strong reference is removed, making weak references unsuitable for such a purpose.

Unowned References

Like a weak reference, an *unowned reference* doesn't keep a strong hold on the instance it refers to. Unlike a weak reference, however, an unowned reference is used when the other instance has the same lifetime or a longer lifetime. You indicate an unowned reference by placing the `unowned` keyword before a property or variable declaration.

Unlike a weak reference, an unowned reference is expected to always have a value. As a result, marking a value as unowned doesn't make it optional, and ARC never sets an unowned reference's value to `nil`.

> IMPORTANT
>
> Use an unowned reference only when you are sure that the reference *always* refers to an instance that hasn't been deallocated.
>
> If you try to access the value of an unowned reference after that instance has been deallocated, you'll get a runtime error.

The following example defines two classes, `Customer` and `CreditCard`, which model a bank customer and a possible credit card for that customer. These two classes each store an instance of the other class as a property. This relationship has the potential to create a strong reference cycle.

The relationship between `Customer` and `CreditCard` is slightly different from the relationship between `Apartment` and `Person` seen in the weak reference example above. In this data model, a customer may or may not have a credit card, but a credit card will *always* be associated with a customer. A `CreditCard` instance never outlives the `Customer` that it refers to. To represent this, the `Customer` class has an optional `card` property, but the `CreditCard` class has an unowned (and non-optional) `customer` property.

Furthermore, a new `CreditCard` instance can *only* be created by passing a `number` value and a `customer` instance to a custom `CreditCard` initializer. This ensures that a `CreditCard` instance always has a `customer` instance associated with it when the `CreditCard` instance is created.

Because a credit card will always have a customer, you define its `customer` property as an unowned reference, to avoid a strong reference cycle:

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { print("\(name) is being deinitialized")
     }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { print("Card #\(number) is being
     deinitialized") }
}
```

> NOTE
>
> The `number` property of the `CreditCard` class is defined with a type of `UInt64` rather than `Int`, to ensure that the `number` property's capacity is large enough to store a 16-digit card number on both 32-bit and 64-bit systems.

This next code snippet defines an optional `Customer` variable called `john`, which will be used to store a reference to a specific customer. This variable

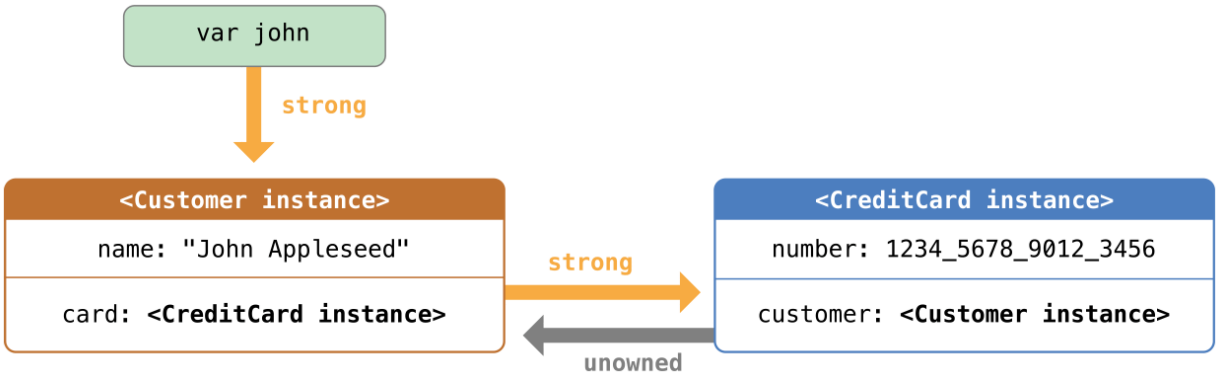
has an initial value of nil, by virtue of being optional:

```
var john: Customer?
```

You can now create a `Customer` instance, and use it to initialize and assign a new `CreditCard` instance as that customer's `card` property:

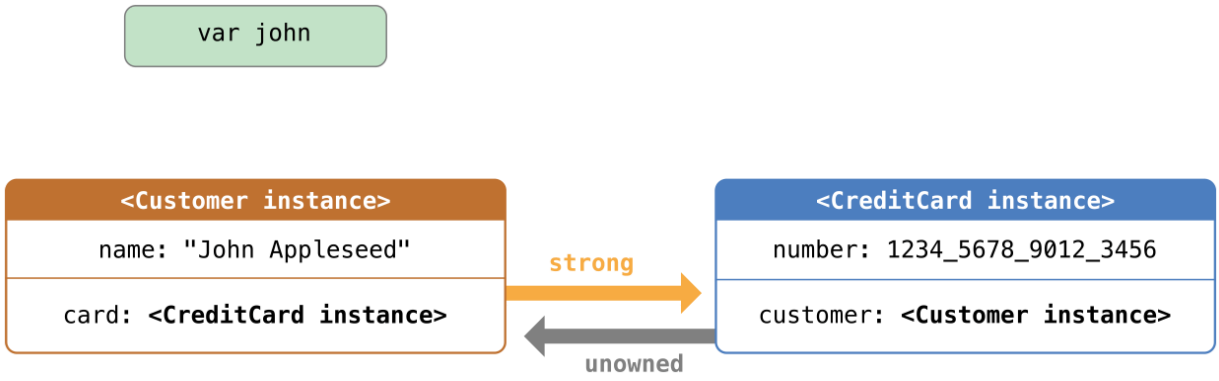
```
1  john = Customer(name: "John Appleseed")
2  john!.card = CreditCard(number: 1234_5678_9012_3456,
       customer: john!)
```

Here's how the references look, now that you've linked the two instances:



The `Customer` instance now has a strong reference to the `CreditCard` instance, and the `CreditCard` instance has an unowned reference to the `Customer` instance.

Because of the unowned `customer` reference, when you break the strong reference held by the `john` variable, there are no more strong references to the `Customer` instance:



Because there are no more strong references to the `Customer` instance, it's deallocated. After this happens, there are no more strong references to the `CreditCard` instance, and it too is deallocated:

```
john = nil
// Prints "John Appleseed is being deinitialized"
// Prints "Card #1234567890123456 is being
     deinitialized"
```

The final code snippet above shows that the deinitializers for the `Customer` instance and `CreditCard` instance both print their "deinitialized" messages after the `john` variable is set to `nil`.

> NOTE
>
> The examples above show how to use *safe* unowned references. Swift also provides *unsafe* unowned references for cases where you need to disable runtime safety checks —for example, for performance reasons. As with all unsafe operations, you take on the responsibility for checking that code for safety.
>
> You indicate an unsafe unowned reference by writing `unowned(unsafe)`. If you try to access an unsafe unowned reference after the instance that it refers to is deallocated, your program will try to access the memory location where the instance used to be, which is an unsafe operation.

Unowned Optional References

You can mark an optional reference to a class as unowned. In terms of the ARC ownership model, an unowned optional reference and a weak reference can both be used in the same contexts. The difference is that when you use an unowned optional reference, you're responsible for making sure it always refers to a valid object or is set to `nil`.

Here's an example that keeps track of the courses offered by a particular department at a school:

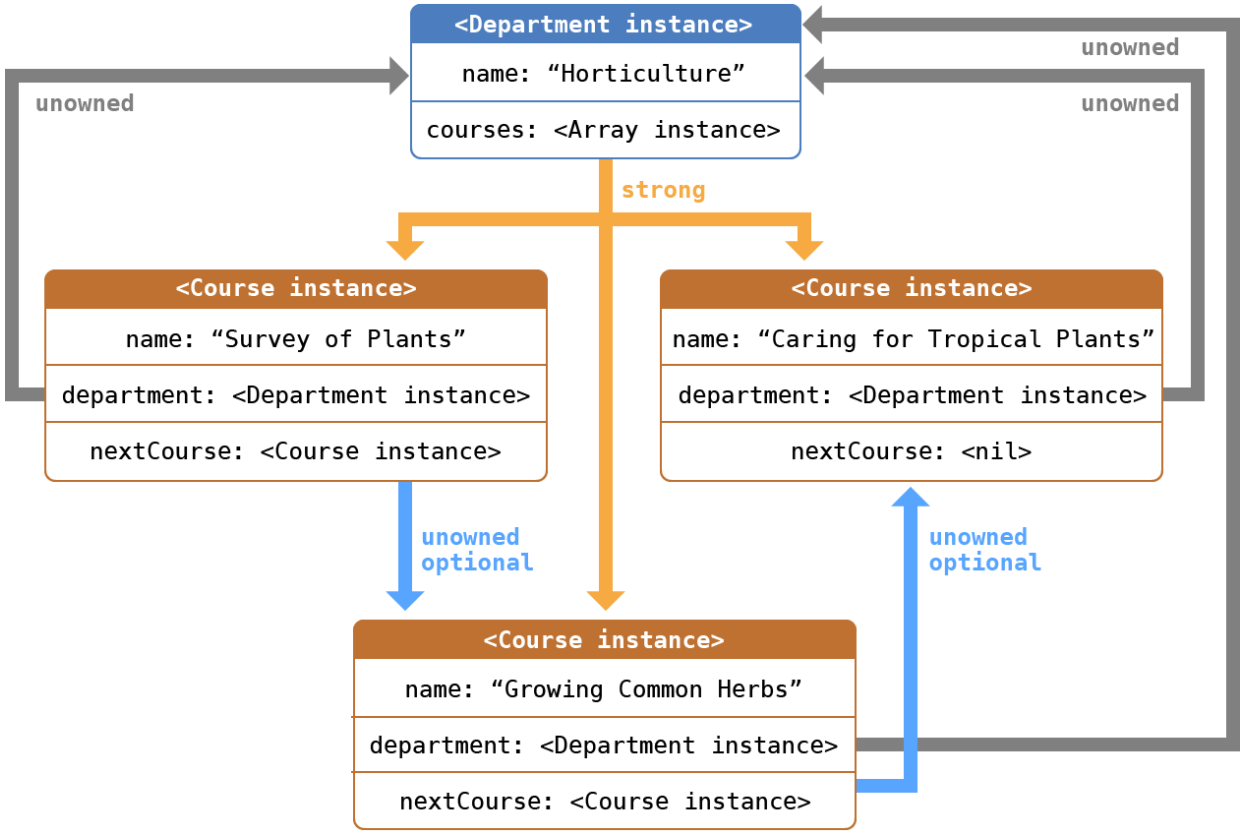
```
class Department {
    var name: String
    var courses: [Course]
    init(name: String) {
        self.name = name
        self.courses = []
    }
}

class Course {
    var name: String
    unowned var department: Department
    unowned var nextCourse: Course?
    init(name: String, in department: Department) {
        self.name = name
        self.department = department
        self.nextCourse = nil
    }
}
```

`Department` maintains a strong reference to each course that the department offers. In the ARC ownership model, a department owns its courses. `Course` has two unowned references, one to the department and one to the next course a student should take; a course doesn't own either of these objects. Every course is part of some department so the `department` property isn't an optional. However, because some courses don't have a recommended follow-on course, the `nextCourse` property is an optional.

Here's an example of using these classes:

```
let department = Department(name: "Horticulture")

let intro = Course(name: "Survey of Plants", in:
     department)
let intermediate = Course(name: "Growing Common
     Herbs", in: department)
let advanced = Course(name: "Caring for Tropical
     Plants", in: department)

intro.nextCourse = intermediate
intermediate.nextCourse = advanced
department.courses = [intro, intermediate, advanced]
```

The code above creates a department and its three courses. The intro and intermediate courses both have a suggested next course stored in their `nextCourse` property, which maintains an unowned optional reference to the course a student should take after completing this one.



An unowned optional reference doesn’t keep a strong hold on the instance of the class that it wraps, and so it doesn’t prevent ARC from deallocating the instance. It behaves the same as an unowned reference does under ARC, except that an unowned optional reference can be `nil`.

Like non-optional unowned references, you’re responsible for ensuring that `nextCourse` always refers to a course that hasn’t been deallocated. In this case, for example, when you delete a course from `department.courses` you also need to remove any references to it that other courses might have.

> NOTE
>
> The underlying type of an optional value is `Optional`, which is an enumeration in the Swift standard library. However, optionals are an exception to the rule that value types can’t be marked with `unowned`.
>
> The optional that wraps the class doesn’t use reference counting, so you don’t need to maintain a strong reference to the optional.

Unowned References and Implicitly Unwrapped Optional Properties
The examples for weak and unowned references above cover two of the more common scenarios in which it's necessary to break a strong reference cycle.

The `Person` and `Apartment` example shows a situation where two properties, both of which are allowed to be `nil`, have the potential to cause a strong reference cycle. This scenario is best resolved with a weak reference.

The `Customer` and `CreditCard` example shows a situation where one property that's allowed to be `nil` and another property that can't be `nil` have the potential to cause a strong reference cycle. This scenario is best resolved with an unowned reference.

However, there's a third scenario, in which *both* properties should always have a value, and neither property should ever be `nil` once initialization is complete. In this scenario, it's useful to combine an unowned property on one class with an implicitly unwrapped optional property on the other class.

This enables both properties to be accessed directly (without optional unwrapping) once initialization is complete, while still avoiding a reference cycle. This section shows you how to set up such a relationship.

The example below defines two classes, `Country` and `City`, each of which stores an instance of the other class as a property. In this data model, every country must always have a capital city, and every city must always belong to a country. To represent this, the `Country` class has a `capitalCity` property, and the `City` class has a `country` property:

```
class Country {
    let name: String
    var capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}

class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

To set up the interdependency between the two classes, the initializer for `City` takes a `Country` instance, and stores this instance in its `country` property.

The initializer for `City` is called from within the initializer for `Country`. However, the initializer for `Country` can't pass `self` to the `City` initializer until a new `Country` instance is fully initialized, as described in [Two-Phase Initialization].

To cope with this requirement, you declare the `capitalCity` property of `Country` as an implicitly unwrapped optional property, indicated by the exclamation point at the end of its type annotation (`City!`). This means that the `capitalCity` property has a default value of `nil`, like any other optional, but can be accessed without the need to unwrap its value as described in [Implicitly Unwrapped Optionals](#).

Because `capitalCity` has a default `nil` value, a new `Country` instance is considered fully initialized as soon as the `Country` instance sets its `name` property within its initializer. This means that the `Country` initializer can start to reference and pass around the implicit `self` property as soon as the `name` property is set. The `Country` initializer can therefore pass `self` as one of the parameters for the `City` initializer when the `Country` initializer is setting its own `capitalCity` property.

All of this means that you can create the `Country` and `City` instances in a single statement, without creating a strong reference cycle, and the `capitalCity` property can be accessed directly, without needing to use an exclamation point to unwrap its optional value:

```
var country = Country(name: "Canada", capitalName:
     "Ottawa")
print("\(country.name)'s capital city is called \
     (country.capitalCity.name)")
// Prints "Canada's capital city is called Ottawa"
```

In the example above, the use of an implicitly unwrapped optional means that all of the two-phase class initializer requirements are satisfied. The `capitalCity` property can be used and accessed like a non-optional value once initialization is complete, while still avoiding a strong reference cycle.

Strong Reference Cycles for Closures

You saw above how a strong reference cycle can be created when two class instance properties hold a strong reference to each other. You also saw how to use weak and unowned references to break these strong reference cycles.

A strong reference cycle can also occur if you assign a closure to a property of a class instance, and the body of that closure captures the instance. This capture might occur because the closure's body accesses a property of the instance, such as `self.someProperty`, or because the closure calls a method on the instance, such as `self.someMethod()`. In either case, these accesses cause the closure to "capture" `self`, creating a strong reference cycle.

This strong reference cycle occurs because closures, like classes, are *reference types*. When you assign a closure to a property, you are assigning a *reference* to that closure. In essence, it's the same problem as above—two strong references are keeping each other alive. However, rather than two class instances, this time it's a class instance and a closure that are keeping each other alive.

Swift provides an elegant solution to this problem, known as a *closure capture list*. However, before you learn how to break a strong reference cycle with a closure capture list, it's useful to understand how such a cycle can be caused.

The example below shows how you can create a strong reference cycle when using a closure that references `self`. This example defines a class called `HTMLElement`, which provides a simple model for an individual element within an HTML document:

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}
```

The `HTMLElement` class defines a `name` property, which indicates the name of the element, such as `"h1"` for a heading element, `"p"` for a paragraph

element, or `"br"` for a line break element. `HTMLElement` also defines an optional `text` property, which you can set to a string that represents the text to be rendered within that HTML element.

In addition to these two simple properties, the `HTMLElement` class defines a lazy property called `asHTML`. This property references a closure that combines `name` and `text` into an HTML string fragment. The `asHTML` property is of type `() -> String`, or "a function that takes no parameters, and returns a `String` value".

By default, the `asHTML` property is assigned a closure that returns a string representation of an HTML tag. This tag contains the optional `text` value if it exists, or no text content if `text` doesn't exist. For a paragraph element, the closure would return `"<p>some text</p>"` or `"<p />"`, depending on whether the `text` property equals `"some text"` or `nil`.

The `asHTML` property is named and used somewhat like an instance method. However, because `asHTML` is a closure property rather than an instance method, you can replace the default value of the `asHTML` property with a custom closure, if you want to change the HTML rendering for a particular HTML element.

For example, the `asHTML` property could be set to a closure that defaults to some text if the `text` property is `nil`, in order to prevent the representation from returning an empty HTML tag:

```
let heading = HTMLElement(name: "h1")
let defaultText = "some default text"
heading.asHTML = {
    return "<\(heading.name)>\(heading.text ??
     defaultText)</\(heading.name)>"
}
print(heading.asHTML())
// Prints "<h1>some default text</h1>"
```

> NOTE
>
> The `asHTML` property is declared as a lazy property, because it's only needed if and when the element actually needs to be rendered as a string value for some HTML output target. The fact that `asHTML` is a lazy property means that you can refer to `self` within the default closure, because the lazy property will not be accessed until after initialization has been completed and `self` is known to exist.

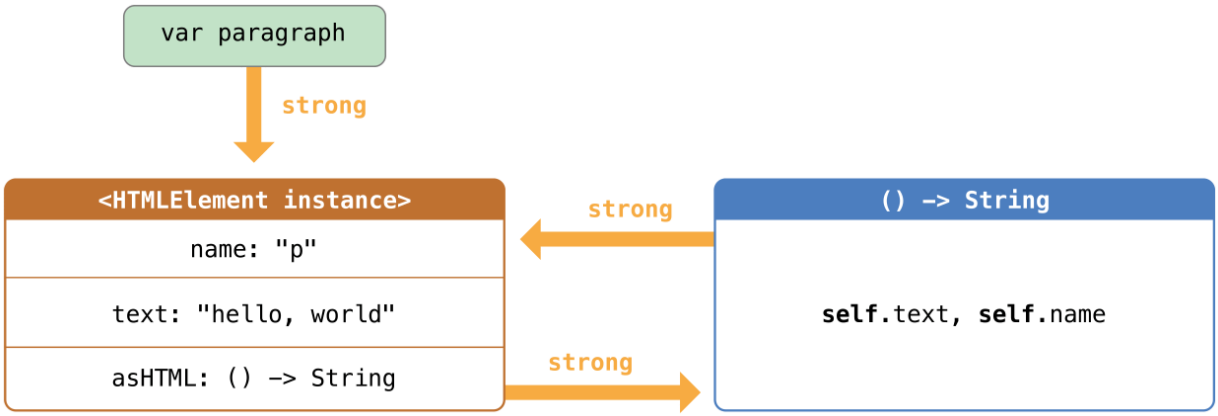
The `HTMLElement` class provides a single initializer, which takes a `name` argument and (if desired) a `text` argument to initialize a new element. The class also defines a deinitializer, which prints a message to show when an `HTMLElement` instance is deallocated.

Here's how you use the `HTMLElement` class to create and print a new instance:

```
var paragraph: HTMLElement? = HTMLElement(name: "p",
     text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

> NOTE
>
> The `paragraph` variable above is defined as an *optional* `HTMLElement`, so that it can be set to `nil` below to demonstrate the presence of a strong reference cycle.

Unfortunately, the `HTMLElement` class, as written above, creates a strong reference cycle between an `HTMLElement` instance and the closure used for its default `asHTML` value. Here's how the cycle looks:



The instance's `asHTML` property holds a strong reference to its closure. However, because the closure refers to `self` within its body (as a way to reference `self.name` and `self.text`), the closure *captures* self, which means that it holds a strong reference back to the `HTMLElement` instance. A strong reference cycle is created between the two. (For more information about capturing values in a closure, see Capturing Values.)

> NOTE
>
> Even though the closure refers to `self` multiple times, it only captures one strong reference to the `HTMLElement` instance.

If you set the `paragraph` variable to `nil` and break its strong reference to the `HTMLElement` instance, neither the `HTMLElement` instance nor its closure are deallocated, because of the strong reference cycle:

```
paragraph = nil
```

Note that the message in the `HTMLElement` deinitializer isn't printed, which shows that the `HTMLElement` instance isn't deallocated.

Resolving Strong Reference Cycles for Closures

You resolve a strong reference cycle between a closure and a class instance by defining a *capture list* as part of the closure's definition. A capture list defines the rules to use when capturing one or more reference types within the closure's body. As with strong reference cycles between two class instances, you declare each captured reference to be a weak or unowned reference rather than a strong reference. The appropriate choice of weak or unowned depends on the relationships between the different parts of your code.

> NOTE
>
> Swift requires you to write `self.someProperty` or `self.someMethod()` (rather than just `someProperty` or `someMethod()`) whenever you refer to a member of `self` within a closure. This helps you remember that it's possible to capture `self` by accident.

Defining a Capture List

Each item in a capture list is a pairing of the `weak` or `unowned` keyword with a reference to a class instance (such as `self`) or a variable initialized with some value (such as `delegate = self.delegate`). These pairings are written within a pair of square braces, separated by commas.

Place the capture list before a closure's parameter list and return type if they're provided:

```
lazy var someClosure = {
    [unowned self, weak delegate = self.delegate]
    (index: Int, stringToProcess: String) -> String
     in
    // closure body goes here
}
```

If a closure doesn't specify a parameter list or return type because they can be inferred from context, place the capture list at the very start of the closure, followed by the `in` keyword:

```
lazy var someClosure = {
    [unowned self, weak delegate = self.delegate] in
    // closure body goes here
}
```

Weak and Unowned References

Define a capture in a closure as an unowned reference when the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time.

Conversely, define a capture as a weak reference when the captured reference may become `nil` at some point in the future. Weak references are always of an optional type, and automatically become `nil` when the instance they reference is deallocated. This enables you to check for their existence within the closure's body.

> NOTE
>
> If the captured reference will never become `nil`, it should always be captured as an unowned reference, rather than a weak reference.

An unowned reference is the appropriate capture method to use to resolve the strong reference cycle in the `HTMLElement` example from Strong Reference Cycles for Closures above. Here's how you write the `HTMLElement` class to avoid the cycle:

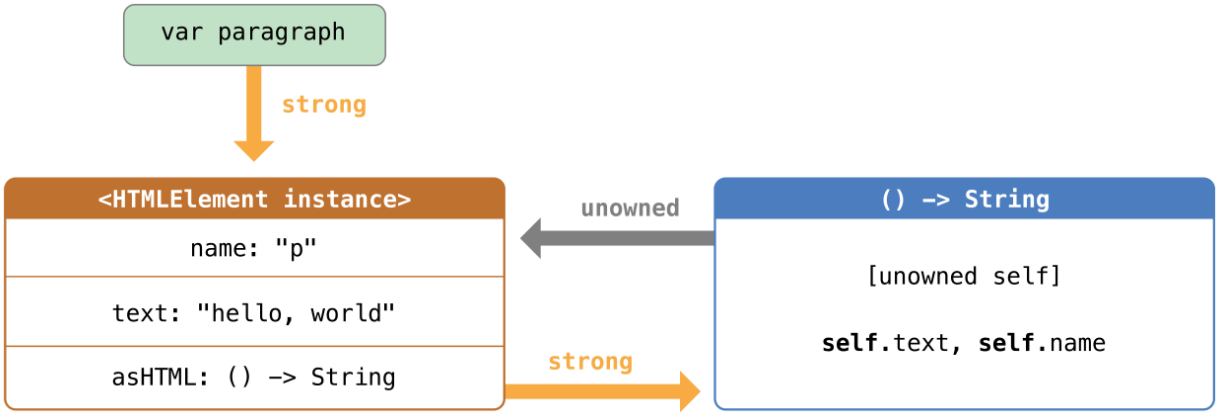
```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        print("\(name) is being deinitialized")
    }

}
```

This implementation of `HTMLElement` is identical to the previous implementation, apart from the addition of a capture list within the `asHTML` closure. In this case, the capture list is `[unowned self]`, which means "capture self as an unowned reference rather than a strong reference".

You can create and print an `HTMLElement` instance as before:

```
var paragraph: HTMLElement? = HTMLElement(name: "p",
     text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

Here's how the references look with the capture list in place:



This time, the capture of `self` by the closure is an unowned reference, and doesn't keep a strong hold on the `HTMLElement` instance it has captured. If you set the strong reference from the `paragraph` variable to `nil`, the `HTMLElement` instance is deallocated, as can be seen from the printing of its deinitializer message in the example below:

```
paragraph = nil
// Prints "p is being deinitialized"
```

For more information about capture lists, see Capture Lists.

Memory Safety

By default, Swift prevents unsafe behavior from happening in your code. For example, Swift ensures that variables are initialized before they're used, memory isn't accessed after it's been deallocated, and array indices are checked for out-of-bounds errors.

Swift also makes sure that multiple accesses to the same area of memory don't conflict, by requiring code that modifies a location in memory to have exclusive access to that memory. Because Swift manages memory automatically, most of the time you don't have to think about accessing memory at all. However, it's important to understand where potential conflicts can occur, so you can avoid writing code that has conflicting access to memory. If your code does contain conflicts, you'll get a compile-time or runtime error.

Understanding Conflicting Access to Memory

Access to memory happens in your code when you do things like set the value of a variable or pass an argument to a function. For example, the following code contains both a read access and a write access:

```
// A write access to the memory where one is stored.
var one = 1

// A read access from the memory where one is stored.
print("We're number \(one)!")
```

A conflicting access to memory can occur when different parts of your code are trying to access the same location in memory at the same time. Multiple accesses to a location in memory at the same time can produce unpredictable or inconsistent behavior. In Swift, there are ways to modify a value that span several lines of code, making it possible to attempt to access a value in the middle of its own modification.

You can see a similar problem by thinking about how you update a budget that's written on a piece of paper. Updating the budget is a two-step process: First you add the items' names and prices, and then you change the total amount to reflect the items currently on the list. Before and after the update, you can read any information from the budget and get a correct answer, as shown in the figure below.

While you're adding items to the budget, it's in a temporary, invalid state because the total amount hasn't been updated to reflect the newly added items. Reading the total amount during the process of adding an item gives you incorrect information.

This example also demonstrates a challenge you may encounter when fixing conflicting access to memory: There are sometimes multiple ways to fix the conflict that produce different answers, and it's not always obvious which answer is correct. In this example, depending on whether you wanted the original total amount or the updated total amount, either $5 or $320 could be the correct answer. Before you can fix the conflicting access, you have to determine what it was intended to do.

> NOTE
>
> If you've written concurrent or multithreaded code, conflicting access to memory might be a familiar problem. However, the conflicting access discussed here can happen on a single thread and *doesn't* involve concurrent or multithreaded code.
>
> If you have conflicting access to memory from within a single thread, Swift guarantees that you'll get an error at either compile time or runtime. For multithreaded code, use Thread Sanitizer to help detect conflicting access across threads.

Characteristics of Memory Access

There are three characteristics of memory access to consider in the context of conflicting access: whether the access is a read or a write, the duration of the access, and the location in memory being accessed. Specifically, a conflict occurs if you have two accesses that meet all of the following conditions:

- At least one is a write access or a nonatomic access.
- They access the same location in memory.
- Their durations overlap.

The difference between a read and write access is usually obvious: a write access changes the location in memory, but a read access doesn't. The location in memory refers to what is being accessed—for example, a variable, constant, or property. The duration of a memory access is either instantaneous or long-term.

An operation is *atomic* if it uses only C atomic operations; otherwise it's nonatomic. For a list of those functions, see the `stdatomic(3)` man page.

An access is *instantaneous* if it's not possible for other code to run after that access starts but before it ends. By their nature, two instantaneous accesses can't happen at the same time. Most memory access is instantaneous. For

example, all the read and write accesses in the code listing below are instantaneous:

```
1  func oneMore(than number: Int) -> Int {
2      return number + 1
3  }
4
5  var myNumber = 1
6  myNumber = oneMore(than: myNumber)
7  print(myNumber)
8  // Prints "2"
```

However, there are several ways to access memory, called *long-term* accesses, that span the execution of other code. The difference between instantaneous access and long-term access is that it's possible for other code to run after a long-term access starts but before it ends, which is called *overlap*. A long-term access can overlap with other long-term accesses and instantaneous accesses.

Overlapping accesses appear primarily in code that uses in-out parameters in functions and methods or mutating methods of a structure. The specific kinds of Swift code that use long-term accesses are discussed in the sections below.

Conflicting Access to In-Out Parameters

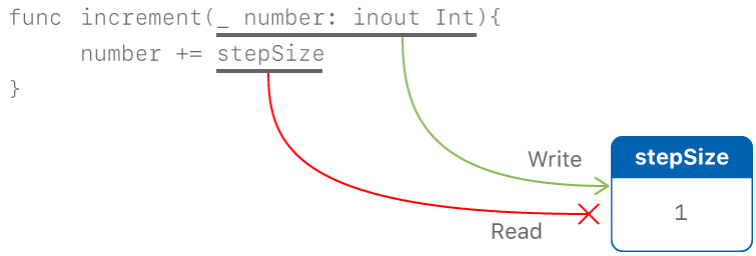
A function has long-term write access to all of its in-out parameters. The write access for an in-out parameter starts after all of the non-in-out parameters have been evaluated and lasts for the entire duration of that

function call. If there are multiple in-out parameters, the write accesses start in the same order as the parameters appear.

One consequence of this long-term write access is that you can't access the original variable that was passed as in-out, even if scoping rules and access control would otherwise permit it—any access to the original creates a conflict. For example:

```
var stepSize = 1

func increment(_ number: inout Int) {
    number += stepSize
}

increment(&stepSize)
// Error: conflicting accesses to stepSize
```

In the code above, `stepSize` is a global variable, and it's normally accessible from within `increment(_:)`. However, the read access to `stepSize` overlaps with the write access to `number`. As shown in the figure below, both `number` and `stepSize` refer to the same location in memory. The read and write accesses refer to the same memory and they overlap, producing a conflict.



One way to solve this conflict is to make an explicit copy of `stepSize`:

```
// Make an explicit copy.
var copyOfStepSize = stepSize
increment(&copyOfStepSize)

// Update the original.
stepSize = copyOfStepSize
// stepSize is now 2
```

When you make a copy of `stepSize` before calling `increment(_:)`, it's clear that the value of `copyOfStepSize` is incremented by the current step size. The read access ends before the write access starts, so there isn't a conflict.

Another consequence of long-term write access to in-out parameters is that passing a single variable as the argument for multiple in-out parameters of the same function produces a conflict. For example:

```
func balance(_ x: inout Int, _ y: inout Int) {
    let sum = x + y
    x = sum / 2
    y = sum - x
}
var playerOneScore = 42
var playerTwoScore = 30
balance(&playerOneScore, &playerTwoScore)  // OK
balance(&playerOneScore, &playerOneScore)
// Error: conflicting accesses to playerOneScore
```

The `balance(_:_:)` function above modifies its two parameters to divide the total value evenly between them. Calling it with `playerOneScore` and `playerTwoScore` as arguments doesn't produce a conflict—there are two write accesses that overlap in time, but they access different locations in memory. In contrast, passing `playerOneScore` as the value for both parameters produces a conflict because it tries to perform two write accesses to the same location in memory at the same time.

> NOTE
>
> Because operators are functions, they can also have long-term accesses to their in-out parameters. For example, if `balance(_:_:)` was an operator function named `<^>`, writing `playerOneScore <^> playerOneScore` would result in the same conflict as `balance(&playerOneScore, &playerOneScore)`.

Conflicting Access to self in Methods

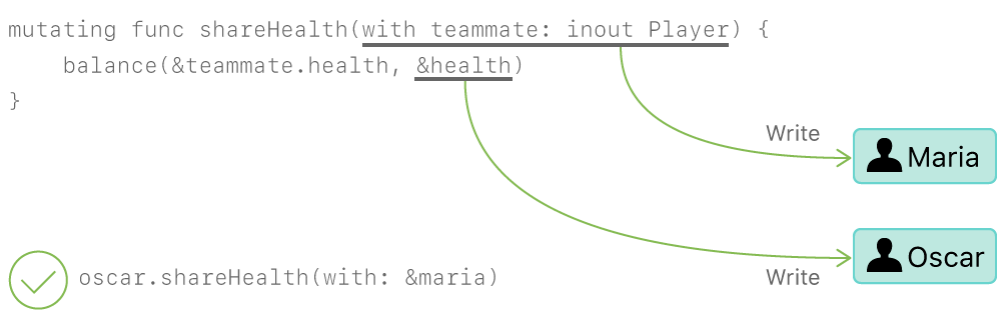
A mutating method on a structure has write access to `self` for the duration of the method call. For example, consider a game where each player has a health amount, which decreases when taking damage, and an energy amount, which decreases when using special abilities.

```swift
struct Player {
    var name: String
    var health: Int
    var energy: Int

    static let maxHealth = 10
    mutating func restoreHealth() {
        health = Player.maxHealth
    }
}
```

In the `restoreHealth()` method above, a write access to `self` starts at the beginning of the method and lasts until the method returns. In this case, there's no other code inside `restoreHealth()` that could have an overlapping access to the properties of a `Player` instance. The `shareHealth(with:)` method below takes another `Player` instance as an in-out parameter, creating the possibility of overlapping accesses.

```
extension Player {
    mutating func shareHealth(with teammate: inout Player) {
        balance(&teammate.health, &health)
    }
}

var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Maria", health: 5, energy: 10)
oscar.shareHealth(with: &maria)  // OK
```

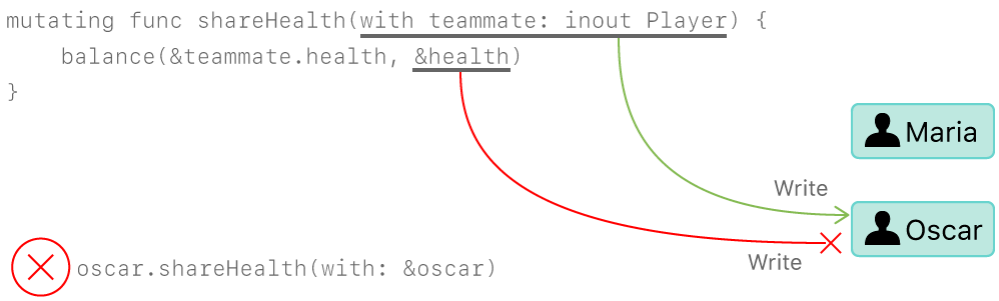
In the example above, calling the `shareHealth(with:)` method for Oscar's player to share health with Maria's player doesn't cause a conflict. There's a write access to `oscar` during the method call because `oscar` is the value of `self` in a mutating method, and there's a write access to `maria` for the same duration because `maria` was passed as an in-out parameter. As shown in the figure below, they access different locations in memory. Even though the two write accesses overlap in time, they don't conflict.



However, if you pass `oscar` as the argument to `shareHealth(with:)`, there's a conflict:

```
oscar.shareHealth(with: &oscar)
// Error: conflicting accesses to oscar
```

The mutating method needs write access to `self` for the duration of the method, and the in-out parameter needs write access to `teammate` for the same duration. Within the method, both `self` and `teammate` refer to the same location in memory—as shown in the figure below. The two write accesses refer to the same memory and they overlap, producing a conflict.



Conflicting Access to Properties

Types like structures, tuples, and enumerations are made up of individual constituent values, such as the properties of a structure or the elements of a tuple. Because these are value types, mutating any piece of the value mutates the whole value, meaning read or write access to one of the properties requires read or write access to the whole value. For example, overlapping write accesses to the elements of a tuple produces a conflict:

```
var playerInformation = (health: 10, energy: 20)
balance(&playerInformation.health,
     &playerInformation.energy)
// Error: conflicting access to properties of
     playerInformation
```

In the example above, calling `balance(_:_:)` on the elements of a tuple produces a conflict because there are overlapping write accesses to `playerInformation`. Both `playerInformation.health` and `playerInformation.energy` are passed as in-out parameters, which means `balance(_:_:)` needs write access to them for the duration of the function call. In both cases, a write access to the tuple element requires a write access to the entire tuple. This means there are two write accesses to `playerInformation` with durations that overlap, causing a conflict.

The code below shows that the same error appears for overlapping write accesses to the properties of a structure that's stored in a global variable.

```
var holly = Player(name: "Holly", health: 10,
     energy: 10)
balance(&holly.health, &holly.energy)  // Error
```

In practice, most access to the properties of a structure can overlap safely. For example, if the variable `holly` in the example above is changed to a local variable instead of a global variable, the compiler can prove that overlapping access to stored properties of the structure is safe:

```
func someFunction() {
    var oscar = Player(name: "Oscar", health: 10,
     energy: 10)
    balance(&oscar.health, &oscar.energy)  // OK
}
```

In the example above, Oscar's health and energy are passed as the two in-out parameters to `balance(_:_:)`. The compiler can prove that memory safety is preserved because the two stored properties don't interact in any way.

The restriction against overlapping access to properties of a structure isn't always necessary to preserve memory safety. Memory safety is the desired guarantee, but exclusive access is a stricter requirement than memory safety—which means some code preserves memory safety, even though it violates exclusive access to memory. Swift allows this memory-safe code if the compiler can prove that the nonexclusive access to memory is still safe. Specifically, it can prove that overlapping access to properties of a structure is safe if the following conditions apply:

- You're accessing only stored properties of an instance, not computed properties or class properties.
- The structure is the value of a local variable, not a global variable.
- The structure is either not captured by any closures, or it's captured only by nonescaping closures.

If the compiler can't prove the access is safe, it doesn't allow the access.

Access Control

Access control restricts access to parts of your code from code in other source files and modules. This feature enables you to hide the implementation details of your code, and to specify a preferred interface through which that code can be accessed and used.

You can assign specific access levels to individual types (classes, structures, and enumerations), as well as to properties, methods, initializers, and subscripts belonging to those types. Protocols can be restricted to a certain context, as can global constants, variables, and functions.

In addition to offering various levels of access control, Swift reduces the need to specify explicit access control levels by providing default access levels for typical scenarios. Indeed, if you are writing a single-target app, you may not need to specify explicit access control levels at all.

> NOTE
>
> The various aspects of your code that can have access control applied to them (properties, types, functions, and so on) are referred to as "entities" in the sections below, for brevity.

Modules and Source Files

Swift's access control model is based on the concept of modules and source files.

A *module* is a single unit of code distribution—a framework or application that's built and shipped as a single unit and that can be imported by another module with Swift's `import` keyword.

Each build target (such as an app bundle or framework) in Xcode is treated as a separate module in Swift. If you group together aspects of your app's code as a stand-alone framework—perhaps to encapsulate and reuse that code across multiple applications—then everything you define within that framework will be part of a separate module when it's imported and used within an app, or when it's used within another framework.

A *source file* is a single Swift source code file within a module (in effect, a single file within an app or framework). Although it's common to define individual types in separate source files, a single source file can contain definitions for multiple types, functions, and so on.

Access Levels

Swift provides five different *access levels* for entities within your code. These access levels are relative to the source file in which an entity is defined, and also relative to the module that source file belongs to.

- *Open access* and *public access* enable entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module. You typically use open or public access when specifying the public interface to a framework. The difference between open and public access is described below.

- *Internal access* enables entities to be used within any source file from their defining module, but not in any source file outside of that module. You typically use internal access when defining an app's or a framework's internal structure.

- *File-private access* restricts the use of an entity to its own defining source file. Use file-private access to hide the implementation details

of a specific piece of functionality when those details are used within an entire file.

- *Private access* restricts the use of an entity to the enclosing declaration, and to extensions of that declaration that are in the same file. Use private access to hide the implementation details of a specific piece of functionality when those details are used only within a single declaration.

Open access is the highest (least restrictive) access level and private access is the lowest (most restrictive) access level.

Open access applies only to classes and class members, and it differs from public access by allowing code outside the module to subclass and override, as discussed below in Subclassing. Marking a class as open explicitly indicates that you've considered the impact of code from other modules using that class as a superclass, and that you've designed your class's code accordingly.

Guiding Principle of Access Levels

Access levels in Swift follow an overall guiding principle: *No entity can be defined in terms of another entity that has a lower (more restrictive) access level*.

For example:

- A public variable can't be defined as having an internal, file-private, or private type, because the type might not be available everywhere that the public variable is used.

- A function can't have a higher access level than its parameter types and return type, because the function could be used in situations where its constituent types are unavailable to the surrounding code.

The specific implications of this guiding principle for different aspects of the language are covered in detail below.

Default Access Levels

All entities in your code (with a few specific exceptions, as described later in this chapter) have a default access level of internal if you don't specify an explicit access level yourself. As a result, in many cases you don't need to specify an explicit access level in your code.

Access Levels for Single-Target Apps

When you write a simple single-target app, the code in your app is typically self-contained within the app and doesn't need to be made available outside of the app's module. The default access level of internal already matches this requirement. Therefore, you don't need to specify a custom access level. You may, however, want to mark some parts of your code as file private or private in order to hide their implementation details from other code within the app's module.

Access Levels for Frameworks

When you develop a framework, mark the public-facing interface to that framework as open or public so that it can be viewed and accessed by other modules, such as an app that imports the framework. This public-facing interface is the application programming interface (or API) for the framework.

> NOTE
>
> Any internal implementation details of your framework can still use the default access level of internal, or can be marked as private or file private if you want to hide them from other parts of the framework's internal code. You need to mark an entity as open or public only if you want it to become part of your framework's API.

Access Levels for Unit Test Targets
When you write an app with a unit test target, the code in your app needs to be made available to that module in order to be tested. By default, only entities marked as open or public are accessible to other modules. However, a unit test target can access any internal entity, if you mark the import declaration for a product module with the `@testable` attribute and compile that product module with testing enabled.

Access Control Syntax

Define the access level for an entity by placing one of the `open`, `public`, `internal`, `fileprivate`, or `private` modifiers at the beginning of the entity's declaration.

```
public class SomePublicClass {}
internal class SomeInternalClass {}
fileprivate class SomeFilePrivateClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
fileprivate func someFilePrivateFunction() {}
private func somePrivateFunction() {}
```

Unless otherwise specified, the default access level is internal, as described in [Default Access Levels]. This means that `SomeInternalClass` and `someInternalConstant` can be written without an explicit access-level modifier, and will still have an access level of internal:

```
class SomeInternalClass {}              // implicitly internal
let someInternalConstant = 0            // implicitly internal
```

Custom Types

If you want to specify an explicit access level for a custom type, do so at the point that you define the type. The new type can then be used wherever its access level permits. For example, if you define a file-private class, that class can only be used as the type of a property, or as a function parameter or return type, in the source file in which the file-private class is defined.

The access control level of a type also affects the default access level of that type's *members* (its properties, methods, initializers, and subscripts). If you define a type's access level as private or file private, the default access level of its members will also be private or file private. If you define a type's access level as internal or public (or use the default access level of internal without specifying an access level explicitly), the default access level of the type's members will be internal.

> IMPORTANT
>
> A public type defaults to having internal members, not public members. If you want a type member to be public, you must explicitly mark it as such. This requirement ensures that the public-facing API for a type is something you opt in to publishing, and avoids presenting the internal workings of a type as public API by mistake.

```
1  public class SomePublicClass {                    //
       explicitly public class
2      public var somePublicProperty = 0              //
       explicitly public class member
3      var someInternalProperty = 0                   //
       implicitly internal class member
4      fileprivate func someFilePrivateMethod() {}  //
       explicitly file-private class member
5      private func somePrivateMethod() {}            //
       explicitly private class member
6  }
7
8  class SomeInternalClass {                         //
       implicitly internal class
9      var someInternalProperty = 0                   //
       implicitly internal class member
10     fileprivate func someFilePrivateMethod() {}  //
       explicitly file-private class member
11     private func somePrivateMethod() {}            //
       explicitly private class member
12 }
13
14 fileprivate class SomeFilePrivateClass {          //
       explicitly file-private class
15     func someFilePrivateMethod() {}                //
       implicitly file-private class member
```

```
    private func somePrivateMethod() {}          // explicitly private class member
}

private class SomePrivateClass {                // explicitly private class
    func somePrivateMethod() {}                  // implicitly private class member
}
```

Tuple Types

The access level for a tuple type is the most restrictive access level of all types used in that tuple. For example, if you compose a tuple from two different types, one with internal access and one with private access, the access level for that compound tuple type will be private.

> NOTE
>
> Tuple types don't have a standalone definition in the way that classes, structures, enumerations, and functions do. A tuple type's access level is determined automatically from the types that make up the tuple type, and can't be specified explicitly.

Function Types

The access level for a function type is calculated as the most restrictive access level of the function's parameter types and return type. You must specify the access level explicitly as part of the function's definition if the function's calculated access level doesn't match the contextual default.

The example below defines a global function called `someFunction()`, without providing a specific access-level modifier for the function itself. You might expect this function to have the default access level of "internal", but this isn't the case. In fact, `someFunction()` won't compile as written below:

```
1  func someFunction() -> (SomeInternalClass,
       SomePrivateClass) {
2      // function implementation goes here
3  }
```

The function's return type is a tuple type composed from two of the custom classes defined above in Custom Types. One of these classes is defined as internal, and the other is defined as private. Therefore, the overall access level of the compound tuple type is private (the minimum access level of the tuple's constituent types).

Because the function's return type is private, you must mark the function's overall access level with the `private` modifier for the function declaration to be valid:

```
1  private func someFunction() -> (SomeInternalClass,
       SomePrivateClass) {
2      // function implementation goes here
3  }
```

It's not valid to mark the definition of `someFunction()` with the `public` or `internal` modifiers, or to use the default setting of internal, because public or internal users of the function might not have appropriate access to the private class used in the function's return type.

Enumeration Types
The individual cases of an enumeration automatically receive the same access level as the enumeration they belong to. You can't specify a different access level for individual enumeration cases.

In the example below, the `CompassPoint` enumeration has an explicit access level of public. The enumeration cases `north`, `south`, `east`, and `west` therefore also have an access level of public:

```
public enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

Raw Values and Associated Values

The types used for any raw values or associated values in an enumeration definition must have an access level at least as high as the enumeration's access level. For example, you can't use a private type as the raw-value type of an enumeration with an internal access level.

Nested Types
The access level of a nested type is the same as its containing type, unless the containing type is public. Nested types defined within a public type have an automatic access level of internal. If you want a nested type within a public type to be publicly available, you must explicitly declare the nested type as public.

Subclassing

You can subclass any class that can be accessed in the current access context and that's defined in the same module as the subclass. You can also subclass any open class that's defined in a different module. A subclass can't have a higher access level than its superclass—for example, you can't write a public subclass of an internal superclass.

In addition, for classes that are defined in the same module, you can override any class member (method, property, initializer, or subscript) that's visible in a certain access context. For classes that are defined in another module, you can override any open class member.

An override can make an inherited class member more accessible than its superclass version. In the example below, class `A` is a public class with a file-private method called `someMethod()`. Class `B` is a subclass of `A`, with a reduced access level of "internal". Nonetheless, class `B` provides an override of `someMethod()` with an access level of "internal", which is *higher* than the original implementation of `someMethod()`:

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {}
}
```

It's even valid for a subclass member to call a superclass member that has lower access permissions than the subclass member, as long as the call to the superclass's member takes place within an allowed access level context (that is, within the same source file as the superclass for a file-private

member call, or within the same module as the superclass for an internal member call):

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {
        super.someMethod()
    }
}
```

Because superclass `A` and subclass `B` are defined in the same source file, it's valid for the `B` implementation of `someMethod()` to call `super.someMethod()`.

Constants, Variables, Properties, and Subscripts

A constant, variable, or property can't be more public than its type. It's not valid to write a public property with a private type, for example. Similarly, a subscript can't be more public than either its index type or return type.

If a constant, variable, property, or subscript makes use of a private type, the constant, variable, property, or subscript must also be marked as `private`:

```
private var privateInstance = SomePrivateClass()
```

Getters and Setters
Getters and setters for constants, variables, properties, and subscripts automatically receive the same access level as the constant, variable, property, or subscript they belong to.

You can give a setter a *lower* access level than its corresponding getter, to restrict the read-write scope of that variable, property, or subscript. You assign a lower access level by writing `fileprivate(set)`, `private(set)`, or `internal(set)` before the `var` or `subscript` introducer.

> NOTE
>
> This rule applies to stored properties as well as computed properties. Even though you don't write an explicit getter and setter for a stored property, Swift still synthesizes an implicit getter and setter for you to provide access to the stored property's backing storage. Use `fileprivate(set)`, `private(set)`, and `internal(set)` to change the access level of this synthesized setter in exactly the same way as for an explicit setter in a computed property.

The example below defines a structure called `TrackedString`, which keeps track of the number of times a string property is modified:

```
struct TrackedString {
    private(set) var numberOfEdits = 0
    var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
}
```

The `TrackedString` structure defines a stored string property called `value`, with an initial value of `""` (an empty string). The structure also defines a stored integer property called `numberOfEdits`, which is used to track the

number of times that `value` is modified. This modification tracking is implemented with a `didSet` property observer on the `value` property, which increments `numberOfEdits` every time the `value` property is set to a new value.

The `TrackedString` structure and the `value` property don't provide an explicit access-level modifier, and so they both receive the default access level of internal. However, the access level for the `numberOfEdits` property is marked with a `private(set)` modifier to indicate that the property's getter still has the default access level of internal, but the property is settable only from within code that's part of the `TrackedString` structure. This enables `TrackedString` to modify the `numberOfEdits` property internally, but to present the property as a read-only property when it's used outside the structure's definition.

If you create a `TrackedString` instance and modify its string value a few times, you can see the `numberOfEdits` property value update to match the number of modifications:

```
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment
     numberOfEdits."
stringToEdit.value += " So will this one."
print("The number of edits is \
     (stringToEdit.numberOfEdits)")
// Prints "The number of edits is 3"
```

Although you can query the current value of the `numberOfEdits` property from within another source file, you can't *modify* the property from another source file. This restriction protects the implementation details of the `TrackedString` edit-tracking functionality, while still providing convenient access to an aspect of that functionality.

Note that you can assign an explicit access level for both a getter and a setter if required. The example below shows a version of the `TrackedString` structure in which the structure is defined with an explicit access level of public. The structure's members (including the `numberOfEdits` property) therefore have an internal access level by default. You can make the structure's `numberOfEdits` property getter public, and its property setter private, by combining the `public` and `private(set)` access-level modifiers:

```
public struct TrackedString {
    public private(set) var numberOfEdits = 0
    public var value: String = "" {
        didSet {
            numberOfEdits += 1
        }
    }
    public init() {}
}
```

Initializers

Custom initializers can be assigned an access level less than or equal to the type that they initialize. The only exception is for required initializers (as defined in Required Initializers). A required initializer must have the same access level as the class it belongs to.

As with function and method parameters, the types of an initializer's parameters can't be more private than the initializer's own access level.

Default Initializers

As described in Default Initializers, Swift automatically provides a *default initializer* without any arguments for any structure or base class that provides default values for all of its properties and doesn't provide at least one initializer itself.

A default initializer has the same access level as the type it initializes, unless that type is defined as `public`. For a type that's defined as `public`, the default initializer is considered internal. If you want a public type to be initializable with a no-argument initializer when used in another module, you must explicitly provide a public no-argument initializer yourself as part of the type's definition.

Default Memberwise Initializers for Structure Types

The default memberwise initializer for a structure type is considered private if any of the structure's stored properties are private. Likewise, if any of the structure's stored properties are file private, the initializer is file private. Otherwise, the initializer has an access level of internal.

As with the default initializer above, if you want a public structure type to be initializable with a memberwise initializer when used in another module, you must provide a public memberwise initializer yourself as part of the type's definition.

Protocols

If you want to assign an explicit access level to a protocol type, do so at the point that you define the protocol. This enables you to create protocols that can only be adopted within a certain access context.

The access level of each requirement within a protocol definition is automatically set to the same access level as the protocol. You can't set a

protocol requirement to a different access level than the protocol it supports. This ensures that all of the protocol's requirements will be visible on any type that adopts the protocol.

> NOTE
>
> If you define a public protocol, the protocol's requirements require a public access level for those requirements when they're implemented. This behavior is different from other types, where a public type definition implies an access level of internal for the type's members.

Protocol Inheritance

If you define a new protocol that inherits from an existing protocol, the new protocol can have at most the same access level as the protocol it inherits from. For example, you can't write a public protocol that inherits from an internal protocol.

Protocol Conformance

A type can conform to a protocol with a lower access level than the type itself. For example, you can define a public type that can be used in other modules, but whose conformance to an internal protocol can only be used within the internal protocol's defining module.

The context in which a type conforms to a particular protocol is the minimum of the type's access level and the protocol's access level. For example, if a type is public, but a protocol it conforms to is internal, the type's conformance to that protocol is also internal.

When you write or extend a type to conform to a protocol, you must ensure that the type's implementation of each protocol requirement has at least the same access level as the type's conformance to that protocol. For example, if a public type conforms to an internal protocol, the type's implementation of each protocol requirement must be at least internal.

> NOTE
> In Swift, as in Objective-C, protocol conformance is global—it isn't possible for a type to conform to a protocol in two different ways within the same program.

Extensions

You can extend a class, structure, or enumeration in any access context in which the class, structure, or enumeration is available. Any type members added in an extension have the same default access level as type members declared in the original type being extended. If you extend a public or internal type, any new type members you add have a default access level of internal. If you extend a file-private type, any new type members you add have a default access level of file private. If you extend a private type, any new type members you add have a default access level of private.

Alternatively, you can mark an extension with an explicit access-level modifier (for example, `private`) to set a new default access level for all members defined within the extension. This new default can still be overridden within the extension for individual type members.

You can't provide an explicit access-level modifier for an extension if you're using that extension to add protocol conformance. Instead, the protocol's own access level is used to provide the default access level for each protocol requirement implementation within the extension.

Private Members in Extensions

Extensions that are in the same file as the class, structure, or enumeration that they extend behave as if the code in the extension had been written as part of the original type's declaration. As a result, you can:

- Declare a private member in the original declaration, and access that member from extensions in the same file.
- Declare a private member in one extension, and access that member from another extension in the same file.
- Declare a private member in an extension, and access that member from the original declaration in the same file.

This behavior means you can use extensions in the same way to organize your code, whether or not your types have private entities. For example, given the following simple protocol:

```
protocol SomeProtocol {
    func doSomething()
}
```

You can use an extension to add protocol conformance, like this:

```
struct SomeStruct {
    private var privateVariable = 12
}

extension SomeStruct: SomeProtocol {
    func doSomething() {
        print(privateVariable)
    }
}
```

Generics

The access level for a generic type or generic function is the minimum of the access level of the generic type or function itself and the access level of any type constraints on its type parameters.

Type Aliases

Any type aliases you define are treated as distinct types for the purposes of access control. A type alias can have an access level less than or equal to the access level of the type it aliases. For example, a private type alias can alias a private, file-private, internal, public, or open type, but a public type alias can't alias an internal, file-private, or private type.

> NOTE
>
> This rule also applies to type aliases for associated types used to satisfy protocol conformances.

Advanced Operators

In addition to the operators described in Basic Operators, Swift provides several advanced operators that perform more complex value manipulation. These include all of the bitwise and bit shifting operators you will be familiar with from C and Objective-C.

Unlike arithmetic operators in C, arithmetic operators in Swift don't overflow by default. Overflow behavior is trapped and reported as an error. To opt in to overflow behavior, use Swift's second set of arithmetic operators that overflow by default, such as the overflow addition operator (&+). All of these overflow operators begin with an ampersand (&).

When you define your own structures, classes, and enumerations, it can be useful to provide your own implementations of the standard Swift operators for these custom types. Swift makes it easy to provide tailored implementations of these operators and to determine exactly what their behavior should be for each type you create.

You're not limited to the predefined operators. Swift gives you the freedom to define your own custom infix, prefix, postfix, and assignment operators, with custom precedence and associativity values. These operators can be used and adopted in your code like any of the predefined operators, and you can even extend existing types to support the custom operators you define.

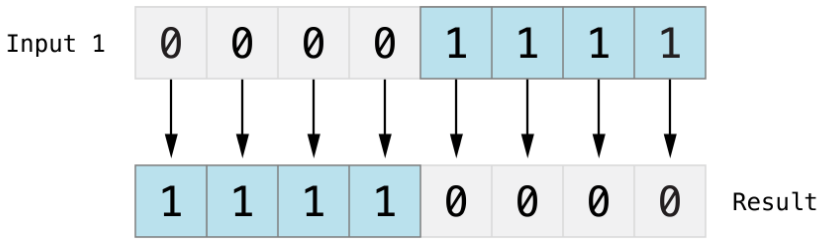
Bitwise Operators

Bitwise operators enable you to manipulate the individual raw data bits within a data structure. They're often used in low-level programming, such as graphics programming and device driver creation. Bitwise operators can also be useful when you work with raw data from external sources, such as encoding and decoding data for communication over a custom protocol.

Swift supports all of the bitwise operators found in C, as described below.

Bitwise NOT Operator

The *bitwise NOT operator* (~) inverts all bits in a number:



The bitwise NOT operator is a prefix operator, and appears immediately before the value it operates on, without any white space:

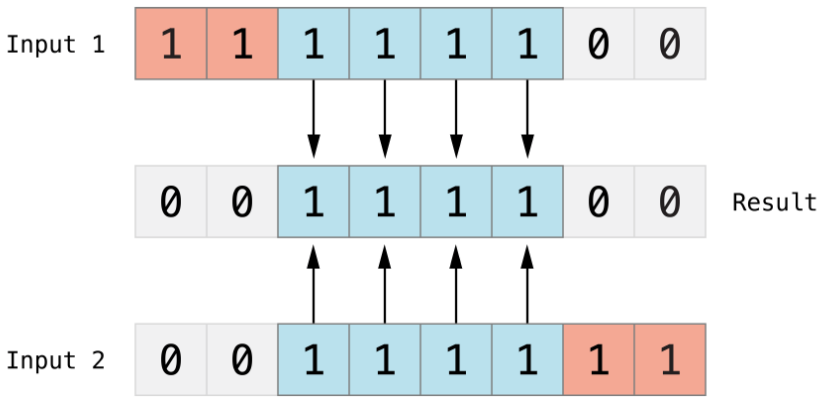
```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // equals 11110000
```

`UInt8` integers have eight bits and can store any value between `0` and `255`. This example initializes a `UInt8` integer with the binary value `00001111`, which has its first four bits set to `0`, and its second four bits set to `1`. This is equivalent to a decimal value of `15`.

The bitwise NOT operator is then used to create a new constant called `invertedBits`, which is equal to `initialBits`, but with all of the bits inverted. Zeros become ones, and ones become zeros. The value of `invertedBits` is `11110000`, which is equal to an unsigned decimal value of `240`.

Bitwise AND Operator

The *bitwise AND operator* (`&`) combines the bits of two numbers. It returns a new number whose bits are set to `1` only if the bits were equal to `1` in *both* input numbers:

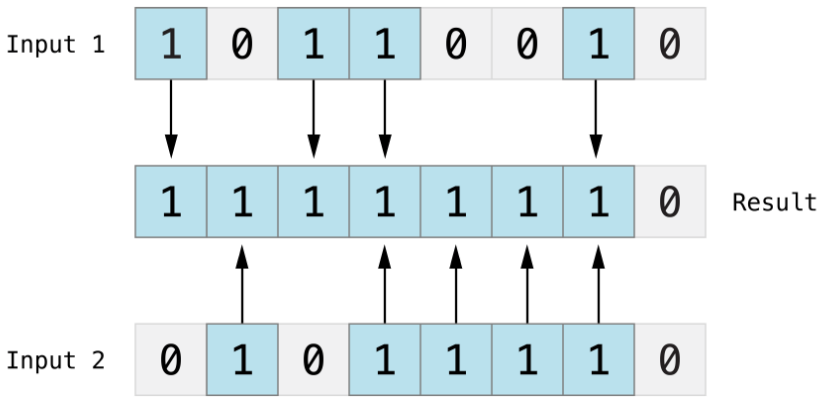


In the example below, the values of `firstSixBits` and `lastSixBits` both have four middle bits equal to `1`. The bitwise AND operator combines them to make the number `00111100`, which is equal to an unsigned decimal value of `60`:

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  //
     equals 00111100
```

Bitwise OR Operator

The *bitwise OR operator* (`|`) compares the bits of two numbers. The operator returns a new number whose bits are set to `1` if the bits are equal to `1` in *either* input number:

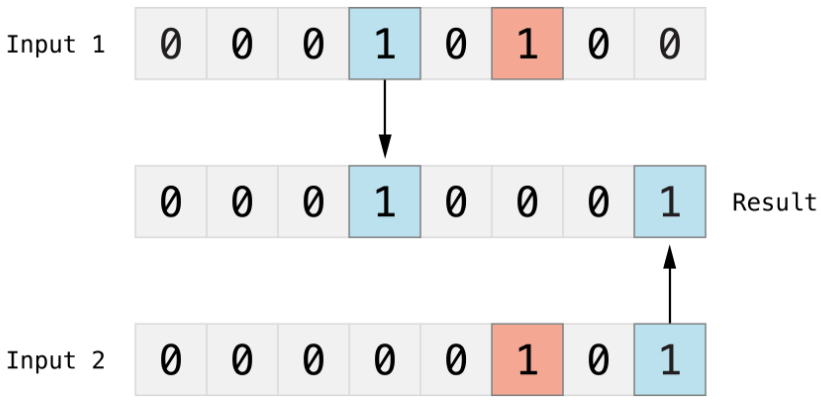


In the example below, the values of `someBits` and `moreBits` have different bits set to `1`. The bitwise OR operator combines them to make the number `11111110`, which equals an unsigned decimal of `254`:

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
let combinedbits = someBits | moreBits  // equals 11111110
```

Bitwise XOR Operator

The *bitwise XOR operator*, or "exclusive OR operator" (^), compares the bits of two numbers. The operator returns a new number whose bits are set to `1` where the input bits are different and are set to `0` where the input bits are the same:



In the example below, the values of `firstBits` and `otherBits` each have a bit set to `1` in a location that the other does not. The bitwise XOR operator sets both of these bits to `1` in its output value. All of the other bits in `firstBits` and `otherBits` match and are set to `0` in the output value:

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
let outputBits = firstBits ^ otherBits  // equals 00010001
```

Bitwise Left and Right Shift Operators

The *bitwise left shift operator* (<<) and *bitwise right shift operator* (>>) move all bits in a number to the left or the right by a certain number of places, according to the rules defined below.

Bitwise left and right shifts have the effect of multiplying or dividing an integer by a factor of two. Shifting an integer's bits to the left by one position doubles its value, whereas shifting it to the right by one position halves its value.

Shifting Behavior for Unsigned Integers

The bit-shifting behavior for unsigned integers is as follows:

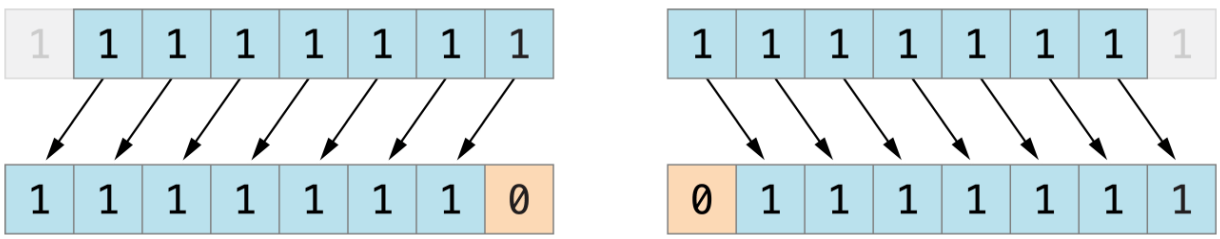
1. Existing bits are moved to the left or right by the requested number of places.

2. Any bits that are moved beyond the bounds of the integer's storage are discarded.

3. Zeros are inserted in the spaces left behind after the original bits are moved to the left or right.

This approach is known as a *logical shift*.

The illustration below shows the results of `11111111 << 1` (which is `11111111` shifted to the left by `1` place), and `11111111 >> 1` (which is `11111111` shifted to the right by `1` place). Blue numbers are shifted, gray numbers are discarded, and orange zeros are inserted:



Here's how bit shifting looks in Swift code:

```swift
let shiftBits: UInt8 = 4   // 00000100 in binary
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
shiftBits >> 2             // 00000001
```

You can use bit shifting to encode and decode values within other data types:

```
1  let pink: UInt32 = 0xCC6699
2  let redComponent = (pink & 0xFF0000) >> 16    //
       redComponent is 0xCC, or 204
3  let greenComponent = (pink & 0x00FF00) >> 8   //
       greenComponent is 0x66, or 102
4  let blueComponent = pink & 0x0000FF           //
       blueComponent is 0x99, or 153
```

This example uses a `UInt32` constant called `pink` to store a Cascading Style Sheets color value for the color pink. The CSS color value `#CC6699` is written as `0xCC6699` in Swift's hexadecimal number representation. This color is then decomposed into its red (`CC`), green (`66`), and blue (`99`) components by the bitwise AND operator (`&`) and the bitwise right shift operator (`>>`).

The red component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0xFF0000`. The zeros in `0xFF0000` effectively "mask" the second and third bytes of `0xCC6699`, causing the `6699` to be ignored and leaving `0xCC0000` as the result.

This number is then shifted 16 places to the right (`>> 16`). Each pair of characters in a hexadecimal number uses 8 bits, so a move 16 places to the right will convert `0xCC0000` into `0x0000CC`. This is the same as `0xCC`, which has a decimal value of `204`.

Similarly, the green component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0x00FF00`, which gives an output value of `0x006600`. This output value is then shifted eight places to the right, giving a value of `0x66`, which has a decimal value of `102`.

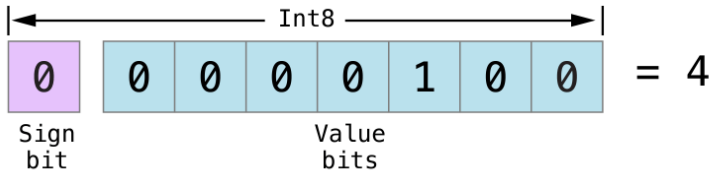
Finally, the blue component is obtained by performing a bitwise AND between the numbers `0xCC6699` and `0x0000FF`, which gives an output value of `0x000099`. Because `0x000099` already equals `0x99`, which has a decimal value of `153`, this value is used without shifting it to the right,

Shifting Behavior for Signed Integers

The shifting behavior is more complex for signed integers than for unsigned integers, because of the way signed integers are represented in binary. (The examples below are based on 8-bit signed integers for simplicity, but the same principles apply for signed integers of any size.)

Signed integers use their first bit (known as the *sign bit*) to indicate whether the integer is positive or negative. A sign bit of `0` means positive, and a sign bit of `1` means negative.

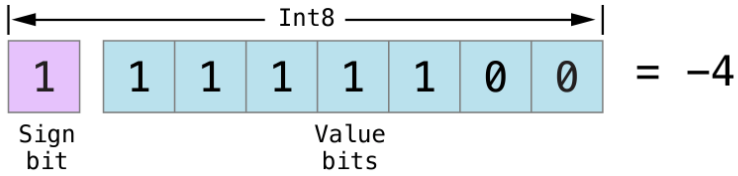
The remaining bits (known as the *value bits*) store the actual value. Positive numbers are stored in exactly the same way as for unsigned integers, counting upwards from `0`. Here's how the bits inside an `Int8` look for the number `4`:



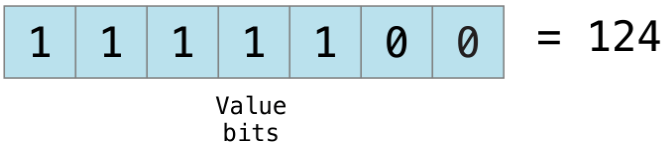
The sign bit is `0` (meaning "positive"), and the seven value bits are just the number `4`, written in binary notation.

Negative numbers, however, are stored differently. They're stored by subtracting their absolute value from `2` to the power of `n`, where `n` is the number of value bits. An eight-bit number has seven value bits, so this means `2` to the power of `7`, or `128`.

Here's how the bits inside an `Int8` look for the number `-4`:

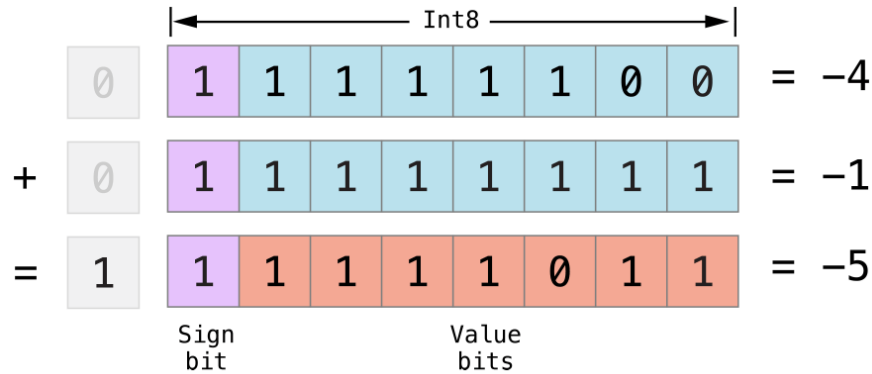


This time, the sign bit is `1` (meaning "negative"), and the seven value bits have a binary value of `124` (which is `128 - 4`):

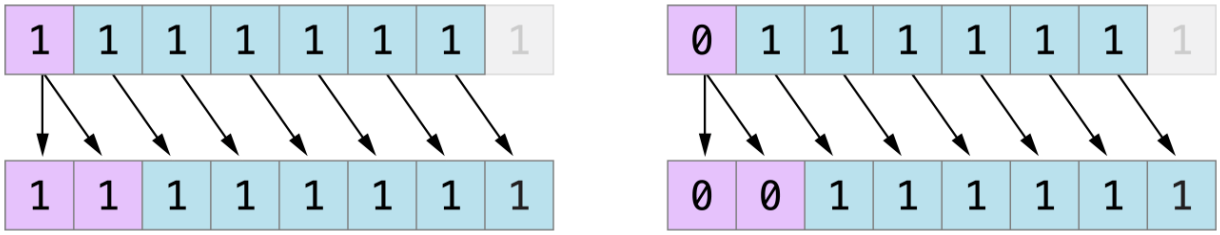


This encoding for negative numbers is known as a *two's complement* representation. It may seem an unusual way to represent negative numbers, but it has several advantages.

First, you can add `-1` to `-4`, simply by performing a standard binary addition of all eight bits (including the sign bit), and discarding anything that doesn't fit in the eight bits once you're done:



Second, the two's complement representation also lets you shift the bits of negative numbers to the left and right like positive numbers, and still end up doubling them for every shift you make to the left, or halving them for every shift you make to the right. To achieve this, an extra rule is used when signed integers are shifted to the right: When you shift signed integers to the right, apply the same rules as for unsigned integers, but fill any empty bits on the left with the *sign bit*, rather than with a zero.



This action ensures that signed integers have the same sign after they're shifted to the right, and is known as an *arithmetic shift*.

Because of the special way that positive and negative numbers are stored, shifting either of them to the right moves them closer to zero. Keeping the sign bit the same during this shift means that negative integers remain negative as their value moves closer to zero.

Overflow Operators

If you try to insert a number into an integer constant or variable that can't hold that value, by default Swift reports an error rather than allowing an invalid value to be created. This behavior gives extra safety when you work with numbers that are too large or too small.

For example, the `Int16` integer type can hold any signed integer between `-32768` and `32767`. Trying to set an `Int16` constant or variable to a number outside of this range causes an error:

```
var potentialOverflow = Int16.max
// potentialOverflow equals 32767, which is the maximum value an Int16 can hold
potentialOverflow += 1
// this causes an error
```

Providing error handling when values get too large or too small gives you much more flexibility when coding for boundary value conditions.

However, when you specifically want an overflow condition to truncate the number of available bits, you can opt in to this behavior rather than triggering an error. Swift provides three arithmetic *overflow operators* that opt in to the overflow behavior for integer calculations. These operators all begin with an ampersand (`&`):

- Overflow addition (`&+`)
- Overflow subtraction (`&-`)
- Overflow multiplication (`&*`)

Value Overflow

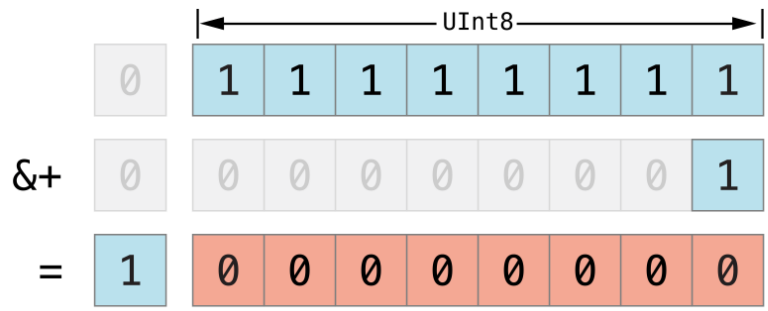
Numbers can overflow in both the positive and negative direction.

Here's an example of what happens when an unsigned integer is allowed to overflow in the positive direction, using the overflow addition operator (`&+`):

```
var unsignedOverflow = UInt8.max
// unsignedOverflow equals 255, which is the maximum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &+ 1
// unsignedOverflow is now equal to 0
```

The variable `unsignedOverflow` is initialized with the maximum value a `UInt8` can hold (`255`, or `11111111` in binary). It's then incremented by `1` using the overflow addition operator (`&+`). This pushes its binary representation just over the size that a `UInt8` can hold, causing it to overflow beyond its bounds, as shown in the diagram below. The value that remains

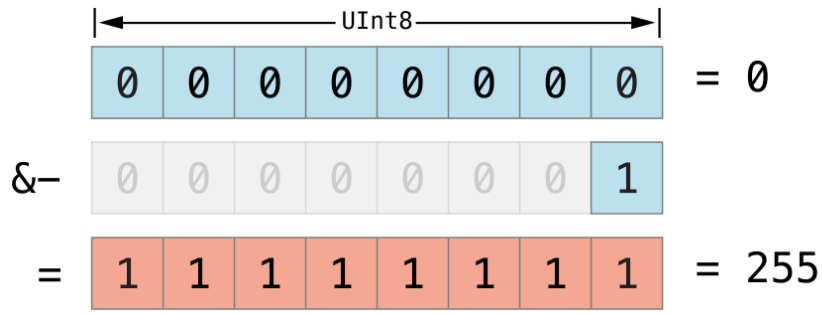
within the bounds of the `UInt8` after the overflow addition is `00000000`, or zero.



Something similar happens when an unsigned integer is allowed to overflow in the negative direction. Here's an example using the overflow subtraction operator (`&-`):

```
var unsignedOverflow = UInt8.min
// unsignedOverflow equals 0, which is the minimum value a UInt8 can hold
unsignedOverflow = unsignedOverflow &- 1
// unsignedOverflow is now equal to 255
```

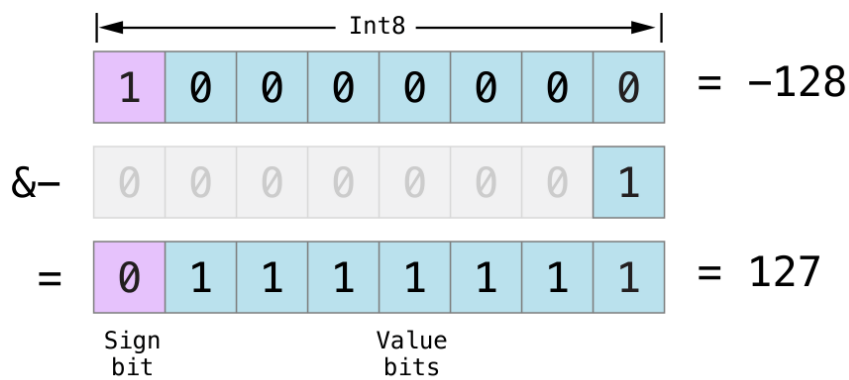
The minimum value that a `UInt8` can hold is zero, or `00000000` in binary. If you subtract `1` from `00000000` using the overflow subtraction operator (`&-`), the number will overflow and wrap around to `11111111`, or `255` in decimal.



Overflow also occurs for signed integers. All addition and subtraction for signed integers is performed in bitwise fashion, with the sign bit included as part of the numbers being added or subtracted, as described in [Bitwise Left and Right Shift Operators](#).

```
var signedOverflow = Int8.min
// signedOverflow equals -128, which is the minimum value an Int8 can hold
signedOverflow = signedOverflow &- 1
// signedOverflow is now equal to 127
```

The minimum value that an `Int8` can hold is `-128`, or `10000000` in binary. Subtracting `1` from this binary number with the overflow operator gives a binary value of `01111111`, which toggles the sign bit and gives positive `127`, the maximum positive value that an `Int8` can hold.



For both signed and unsigned integers, overflow in the positive direction wraps around from the maximum valid integer value back to the minimum, and overflow in the negative direction wraps around from the minimum value to the maximum.

Precedence and Associativity

Operator *precedence* gives some operators higher priority than others; these operators are applied first.

Operator *associativity* defines how operators of the same precedence are grouped together—either grouped from the left, or grouped from the right. Think of it as meaning "they associate with the expression to their left," or "they associate with the expression to their right."

It's important to consider each operator's precedence and associativity when working out the order in which a compound expression will be calculated. For example, operator precedence explains why the following expression equals `17`.

```
2 + 3 % 4 * 5
// this equals 17
```

If you read strictly from left to right, you might expect the expression to be calculated as follows:

- `2` plus `3` equals `5`
- `5` remainder `4` equals `1`
- `1` times `5` equals `5`

However, the actual answer is `17`, not `5`. Higher-precedence operators are evaluated before lower-precedence ones. In Swift, as in C, the remainder operator (`%`) and the multiplication operator (`*`) have a higher precedence than the addition operator (`+`). As a result, they're both evaluated before the addition is considered.

However, remainder and multiplication have the *same* precedence as each other. To work out the exact evaluation order to use, you also need to consider their associativity. Remainder and multiplication both associate with the expression to their left. Think of this as adding implicit parentheses around these parts of the expression, starting from their left:

```
2 + ((3 % 4) * 5)
```

`(3 % 4)` is `3`, so this is equivalent to:

```
2 + (3 * 5)
```

`(3 * 5)` is `15`, so this is equivalent to:

```
2 + 15
```

This calculation yields the final answer of `17`.

For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see Operator Declarations.

> NOTE
>
> Swift's operator precedences and associativity rules are simpler and more predictable than those found in C and Objective-C. However, this means that they aren't exactly the same as in C-based languages. Be careful to ensure that operator interactions still behave in the way you intend when porting existing code to Swift.

Operator Methods

Classes and structures can provide their own implementations of existing operators. This is known as *overloading* the existing operators.

The example below shows how to implement the arithmetic addition operator (`+`) for a custom structure. The arithmetic addition operator is a *binary operator* because it operates on two targets and is said to be *infix* because it appears in between those two targets.

The example defines a `Vector2D` structure for a two-dimensional position vector `(x, y)`, followed by a definition of an *operator method* to add together instances of the `Vector2D` structure:

```
struct Vector2D {
    var x = 0.0, y = 0.0
}

extension Vector2D {
    static func + (left: Vector2D, right: Vector2D) -
     > Vector2D {
        return Vector2D(x: left.x + right.x, y:
     left.y + right.y)
    }
}
```

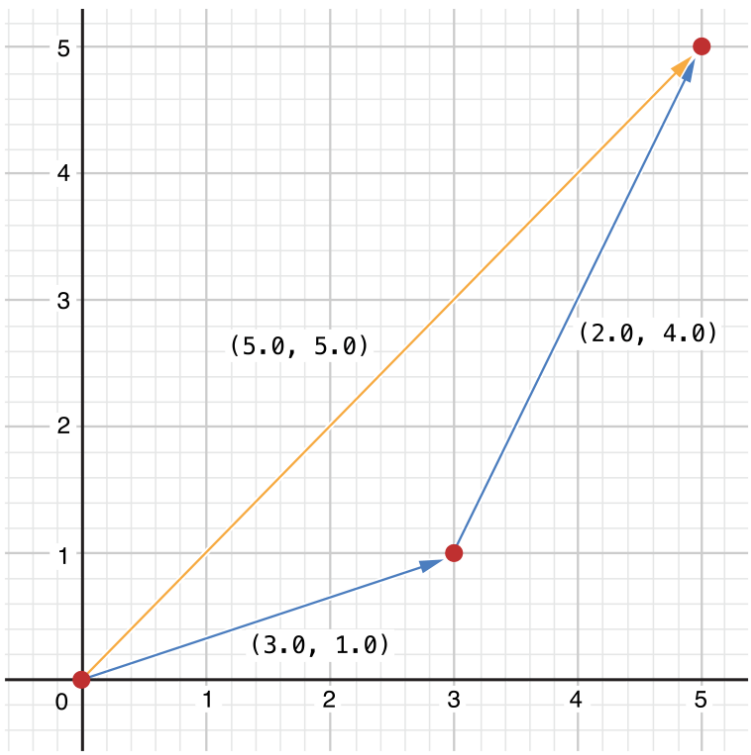
The operator method is defined as a type method on `Vector2D`, with a method name that matches the operator to be overloaded (`+`). Because addition isn't part of the essential behavior for a vector, the type method is defined in an extension of `Vector2D` rather than in the main structure declaration of `Vector2D`. Because the arithmetic addition operator is a binary operator, this operator method takes two input parameters of type `Vector2D` and returns a single output value, also of type `Vector2D`.

In this implementation, the input parameters are named `left` and `right` to represent the `Vector2D` instances that will be on the left side and right side of the `+` operator. The method returns a new `Vector2D` instance, whose `x` and `y` properties are initialized with the sum of the `x` and `y` properties from the two `Vector2D` instances that are added together.

The type method can be used as an infix operator between existing `Vector2D` instances:

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// combinedVector is a Vector2D instance with values
     of (5.0, 5.0)
```

This example adds together the vectors `(3.0, 1.0)` and `(2.0, 4.0)` to make the vector `(5.0, 5.0)`, as illustrated below.



Prefix and Postfix Operators

The example shown above demonstrates a custom implementation of a binary infix operator. Classes and structures can also provide implementations of the standard *unary operators*. Unary operators operate on a single target. They're *prefix* if they precede their target (such as `-a`) and *postfix* operators if they follow their target (such as `b!`).

You implement a prefix or postfix unary operator by writing the `prefix` or `postfix` modifier before the `func` keyword when declaring the operator method:

```
extension Vector2D {
    static prefix func - (vector: Vector2D) -> Vector2D {
        return Vector2D(x: -vector.x, y: -vector.y)
    }
}
```

The example above implements the unary minus operator (`-a`) for `Vector2D` instances. The unary minus operator is a prefix operator, and so this method has to be qualified with the `prefix` modifier.

For simple numeric values, the unary minus operator converts positive numbers into their negative equivalent and vice versa. The corresponding implementation for `Vector2D` instances performs this operation on both the `x` and `y` properties:

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative is a Vector2D instance with values of (-3.0, -4.0)
let alsoPositive = -negative
// alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

Compound Assignment Operators

Compound assignment operators combine assignment (=) with another operation. For example, the addition assignment operator (+=) combines addition and assignment into a single operation. You mark a compound assignment operator's left input parameter type as `inout`, because the parameter's value will be modified directly from within the operator method.

The example below implements an addition assignment operator method for `Vector2D` instances:

```
extension Vector2D {
    static func += (left: inout Vector2D, right:
     Vector2D) {
        left = left + right
    }
}
```

Because an addition operator was defined earlier, you don't need to reimplement the addition process here. Instead, the addition assignment operator method takes advantage of the existing addition operator method, and uses it to set the left value to be the left value plus the right value:

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original now has values of (4.0, 6.0)
```

> NOTE
>
> It isn't possible to overload the default assignment operator (=). Only the compound assignment operators can be overloaded. Similarly, the ternary conditional operator (`a ? b : c`) can't be overloaded.

Equivalence Operators

By default, custom classes and structures don't have an implementation of the *equivalence operators*, known as the *equal to* operator (`==`) and *not equal to* operator (`!=`). You usually implement the `==` operator, and use the standard library's default implementation of the `!=` operator that negates the result of the `==` operator. There are two ways to implement the `==` operator: You can implement it yourself, or for many types, you can ask Swift to synthesize an implementation for you. In both cases, you add conformance to the standard library's `Equatable` protocol.

You provide an implementation of the `==` operator in the same way as you implement other infix operators:

```
extension Vector2D: Equatable {
    static func == (left: Vector2D, right: Vector2D) -> Bool {
        return (left.x == right.x) && (left.y == right.y)
    }
}
```

The example above implements an `==` operator to check whether two `Vector2D` instances have equivalent values. In the context of `Vector2D`, it makes sense to consider "equal" as meaning "both instances have the same `x` values and `y` values", and so this is the logic used by the operator implementation.

You can now use this operator to check whether two `Vector2D` instances are equivalent:

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    print("These two vectors are equivalent.")
}
// Prints "These two vectors are equivalent."
```

In many simple cases, you can ask Swift to provide synthesized implementations of the equivalence operators for you, as described in Adopting a Protocol Using a Synthesized Implementation.

Custom Operators

You can declare and implement your own *custom operators* in addition to the standard operators provided by Swift. For a list of characters that can be used to define custom operators, see Operators.

New operators are declared at a global level using the `operator` keyword, and are marked with the `prefix`, `infix` or `postfix` modifiers:

```
prefix operator +++
```

The example above defines a new prefix operator called `+++`. This operator doesn't have an existing meaning in Swift, and so it's given its own custom meaning below in the specific context of working with `Vector2D` instances. For the purposes of this example, `+++` is treated as a new "prefix doubling" operator. It doubles the `x` and `y` values of a `Vector2D` instance, by adding the vector to itself with the addition assignment operator defined earlier. To implement the `+++` operator, you add a type method called `+++` to `Vector2D` as follows:

```
extension Vector2D {
    static prefix func +++ (vector: inout Vector2D) -
     > Vector2D {
        vector += vector
        return vector
    }
}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

Precedence for Custom Infix Operators

Custom infix operators each belong to a precedence group. A precedence group specifies an operator's precedence relative to other infix operators, as well as the operator's associativity. See Precedence and Associativity for an explanation of how these characteristics affect an infix operator's interaction with other infix operators.

A custom infix operator that isn't explicitly placed into a precedence group is given a default precedence group with a precedence immediately higher than the precedence of the ternary conditional operator.

The following example defines a new custom infix operator called `+-`, which belongs to the precedence group `AdditionPrecedence`:

```
infix operator +-: AdditionPrecedence
extension Vector2D {
    static func +- (left: Vector2D, right: Vector2D) -> Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y - right.y)
    }
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

This operator adds together the `x` values of two vectors, and subtracts the `y` value of the second vector from the first. Because it's in essence an "additive" operator, it has been given the same precedence group as additive infix operators such as `+` and `-`. For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see Operator Declarations. For more information about precedence groups and to see the syntax for defining your own operators and precedence groups, see Operator Declaration.

> NOTE
>
> You don't specify a precedence when defining a prefix or postfix operator. However, if you apply both a prefix and a postfix operator to the same operand, the postfix operator is applied first.

Result Builders

A *result builder* is a type you define that adds syntax for creating nested data, like a list or tree, in a natural, declarative way. The code that uses the result builder can include ordinary Swift syntax, like `if` and `for`, to handle conditional or repeated pieces of data.

The code below defines a few types for drawing on a single line using stars and text.

```
1  protocol Drawable {
2      func draw() -> String
3  }
4  struct Line: Drawable {
5      var elements: [Drawable]
6      func draw() -> String {
7          return elements.map { $0.draw()
       }.joined(separator: "")
8      }
9  }
10 struct Text: Drawable {
11     var content: String
12     init(_ content: String) { self.content = content
       }
13     func draw() -> String { return content }
14 }
15 struct Space: Drawable {
16     func draw() -> String { return " " }
17 }
18 struct Stars: Drawable {
19     var length: Int
20     func draw() -> String { return String(repeating:
       "*", count: length) }
21 }
22 struct AllCaps: Drawable {
23     var content: Drawable
```

```
    func draw() -> String { return content.draw().uppercased() }
}
```

The `Drawable` protocol defines the requirement for something that can be drawn, like a line or shape: The type must implement a `draw()` method. The `Line` structure represents a single-line drawing, and it serves the top-level container for most drawings. To draw a `Line`, the structure calls `draw()` on each of the line's components, and then concatenates the resulting strings into a single string. The `Text` structure wraps a string to make it part of a drawing. The `AllCaps` structure wraps and modifies another drawing, converting any text in the drawing to uppercase.

It's possible to make a drawing with these types by calling their initializers:

```
let name: String? = "Ravi Patel"
let manualDrawing = Line(elements: [
    Stars(length: 3),
    Text("Hello"),
    Space(),
    AllCaps(content: Text((name ?? "World") + "!")),
    Stars(length: 2),
    ])
print(manualDrawing.draw())
// Prints "***Hello RAVI PATEL!**"
```

This code works, but it's a little awkward. The deeply nested parentheses after `AllCaps` are hard to read. The fallback logic to use "World" when `name` is `nil` has to be done inline using the `??` operator, which would be difficult with anything more complex. If you needed to include switches or `for` loops

to build up part of the drawing, there's no way to do that. A result builder lets you rewrite code like this so that it looks like normal Swift code.

To define a result builder, you write the `@resultBuilder` attribute on a type declaration. For example, this code defines a result builder called `DrawingBuilder`, which lets you use a declarative syntax to describe a drawing:

```
@resultBuilder
struct DrawingBuilder {
    static func buildBlock(_ components: Drawable...) -> Drawable {
        return Line(elements: components)
    }
    static func buildEither(first: Drawable) -> Drawable {
        return first
    }
    static func buildEither(second: Drawable) -> Drawable {
        return second
    }
}
```

The `DrawingBuilder` structure defines three methods that implement parts of the result builder syntax. The `buildBlock(_:)` method adds support for writing a series of lines in a block of code. It combines the components in that block into a `Line`. The `buildEither(first:)` and `buildEither(second:)` methods add support for `if-else`.

You can apply the `@DrawingBuilder` attribute to a function's parameter, which turns a closure passed to the function into the value that the result builder creates from that closure. For example:

```
1  func draw(@DrawingBuilder content: () -> Drawable) ->
       Drawable {
2      return content()
3  }
4  func caps(@DrawingBuilder content: () -> Drawable) ->
       Drawable {
5      return AllCaps(content: content())
6  }
7
8  func makeGreeting(for name: String? = nil) ->
       Drawable {
9      let greeting = draw {
10         Stars(length: 3)
11         Text("Hello")
12         Space()
13         caps {
14             if let name = name {
15                 Text(name + "!")
16             } else {
17                 Text("World!")
18             }
19         }
20         Stars(length: 2)
21     }
22     return greeting
23 }
```

```
24  let genericGreeting = makeGreeting()
25  print(genericGreeting.draw())
26  // Prints "***Hello WORLD!**"
27
28  let personalGreeting = makeGreeting(for: "Ravi
        Patel")
29  print(personalGreeting.draw())
30  // Prints "***Hello RAVI PATEL!**"
```

The `makeGreeting(for:)` function takes a `name` parameter and uses it to draw a personalized greeting. The `draw(_:)` and `caps(_:)` functions both take a single closure as their argument, which is marked with the `@DrawingBuilder` attribute. When you call those functions, you use the special syntax that `DrawingBuilder` defines. Swift transforms that declarative description of a drawing into a series of calls to the methods on `DrawingBuilder` to build up the value that's passed as the function argument. For example, Swift transforms the call to `caps(_:)` in that example into code like the following:

```
let capsDrawing = caps {
    let partialDrawing: Drawable
    if let name = name {
        let text = Text(name + "!")
        partialDrawing =
     DrawingBuilder.buildEither(first: text)
    } else {
        let text = Text("World!")
        partialDrawing =
     DrawingBuilder.buildEither(second: text)
    }
    return partialDrawing
}
```

Swift transforms the `if-else` block into calls to the `buildEither(first:)` and `buildEither(second:)` methods. Although you don't call these methods in your own code, showing the result of the transformation makes it easier to see how Swift transforms your code when you use the `DrawingBuilder` syntax.

To add support for writing `for` loops in the special drawing syntax, add a `buildArray(_:)` method.

```swift
extension DrawingBuilder {
    static func buildArray(_ components: [Drawable])
     -> Drawable {
        return Line(elements: components)
    }
}
let manyStars = draw {
    Text("Stars:")
    for length in 1...3 {
        Space()
        Stars(length: length)
    }
}
```

In the code above, the `for` loop creates an array of drawings, and the `buildArray(_:)` method turns that array into a `Line`.

For a complete list of how Swift transforms builder syntax into calls to the builder type's methods, see resultBuilder.

Language Reference

Converted by Evan at Apps Dissected -www.appsdissected.com

About the Language Reference

This part of the book describes the formal grammar of the Swift programming language. The grammar described here is intended to help you understand the language in more detail, rather than to allow you to directly implement a parser or compiler.

The Swift language is relatively small, because many common types, functions, and operators that appear virtually everywhere in Swift code are actually defined in the Swift standard library. Although these types, functions, and operators aren't part of the Swift language itself, they're used extensively in the discussions and code examples in this part of the book.

How to Read the Grammar

The notation used to describe the formal grammar of the Swift programming language follows a few conventions:

- An arrow (→) is used to mark grammar productions and can be read as "can consist of."
- Syntactic categories are indicated by *italic* text and appear on both sides of a grammar production rule.
- Literal words and punctuation are indicated by boldface `constant width` text and appear only on the right-hand side of a grammar production rule.
- Alternative grammar productions are separated by vertical bars (|). When alternative productions are too long to read easily, they're broken into multiple grammar production rules on new lines.

Converted by Evan at Apps Dissected -www.appsdissected.com

- In a few cases, regular font text is used to describe the right-hand side of a grammar production rule.
- Optional syntactic categories and literals are marked by a trailing subscript, *opt*.

As an example, the grammar of a getter-setter block is defined as follows:

```
GRAMMAR OF A GETTER-SETTER BLOCK
getter-setter-block → { getter-clause setter-clause opt } | { setter-
    clause getter-clause }
```

This definition indicates that a getter-setter block can consist of a getter clause followed by an optional setter clause, enclosed in braces, *or* a setter clause followed by a getter clause, enclosed in braces. The grammar production above is equivalent to the following two productions, where the alternatives are spelled out explicitly:

```
GRAMMAR OF A GETTER-SETTER BLOCK
getter-setter-block → { getter-clause setter-clause opt }
getter-setter-block → { setter-clause getter-clause }
```

Lexical Structure

The *lexical structure* of Swift describes what sequence of characters form valid tokens of the language. These valid tokens form the lowest-level building blocks of the language and are used to describe the rest of the language in subsequent chapters. A token consists of an identifier, keyword, punctuation, literal, or operator.

In most cases, tokens are generated from the characters of a Swift source file by considering the longest possible substring from the input text, within the constraints of the grammar that are specified below. This behavior is referred to as *longest match* or *maximal munch*.

Whitespace and Comments

Whitespace has two uses: to separate tokens in the source file and to distinguish between prefix, postfix, and binary operators (see Operators), but is otherwise ignored. The following characters are considered whitespace: space (U+0020), line feed (U+000A), carriage return (U+000D), horizontal tab (U+0009), vertical tab (U+000B), form feed (U+000C) and null (U+0000).

Comments are treated as whitespace by the compiler. Single line comments begin with `//` and continue until a line feed (U+000A) or carriage return (U+000D). Multiline comments begin with `/*` and end with `*/`. Nesting multiline comments is allowed, but the comment markers must be balanced.

Comments can contain additional formatting and markup, as described in Markup Formatting Reference.

GRAMMAR OF WHITESPACE

whitespace → *whitespace-item* *whitespace* *opt*
whitespace-item → *line-break*
whitespace-item → *inline-space*
whitespace-item → *comment*
whitespace-item → *multiline-comment*
whitespace-item → U+0000, U+000B, or U+000C
line-break → U+000A
line-break → U+000D
line-break → U+000D followed by U+000A
inline-spaces → *inline-space* *inline-spaces* *opt*
inline-space → U+0009 or U+0020
comment → `//` *comment-text* *line-break*
multiline-comment → `/*` *multiline-comment-text* `*/`
comment-text → *comment-text-item* *comment-text* *opt*
comment-text-item → Any Unicode scalar value except U+000A or U+000D
multiline-comment-text → *multiline-comment-text-item* *multiline-comment-text* *opt*
multiline-comment-text-item → *multiline-comment*
multiline-comment-text-item → *comment-text-item*
multiline-comment-text-item → Any Unicode scalar value except `/*` or `*/`

Identifiers

Identifiers begin with an uppercase or lowercase letter A through Z, an underscore (_), a noncombining alphanumeric Unicode character in the Basic Multilingual Plane, or a character outside the Basic Multilingual Plane that isn't in a Private Use Area. After the first character, digits and combining Unicode characters are also allowed.

Treat identifiers that begin with an underscore as internal, even if their declaration has the `public` access-level modifier. This convention lets framework authors mark part of an API that clients must not interact with or depend on, even though some limitation requires the declaration to be public. In addition, identifiers that begin with two underscores are reserved for the Swift compiler and standard library.

To use a reserved word as an identifier, put a backtick (`) before and after it. For example, `class` isn't a valid identifier, but `` `class` `` is valid. The backticks aren't considered part of the identifier; `` `x` `` and `x` have the same meaning.

Inside a closure with no explicit parameter names, the parameters are implicitly named `$0`, `$1`, `$2`, and so on. These names are valid identifiers within the scope of the closure.

The compiler synthesizes identifiers that begin with a dollar sign (`$`) for properties that have a property wrapper projection. Your code can interact with these identifiers, but you can't declare identifiers with that prefix. For more information, see the propertyWrapper section of the Attributes chapter.

GRAMMAR OF AN IDENTIFIER

identifier → *identifier-head* *identifier-characters* *opt*
identifier → ` *identifier-head* *identifier-characters* *opt* `
identifier → *implicit-parameter-name*
identifier → *property-wrapper-projection*
identifier-list → *identifier* | *identifier* **,** *identifier-list*
identifier-head → Upper- or lowercase letter A through Z
identifier-head → _
identifier-head → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or U+00B7–U+00BA
identifier-head → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, or U+00F8–U+00FF
identifier-head → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, or U+180F–U+1DBF
identifier-head → U+1E00–U+1FFF
identifier-head → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040, U+2054, or U+2060–U+206F
identifier-head → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, or U+2776–U+2793
identifier-head → U+2C00–U+2DFF or U+2E80–U+2FFF
identifier-head → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, or U+3040–U+D7FF
identifier-head → U+F900–U+FD3D, U+FD40–U+FDCF, U+FDF0–U+FE1F, or U+FE30–U+FE44
identifier-head → U+FE47–U+FFFD
identifier-head → U+10000–U+1FFFD, U+20000–U+2FFFD, U+30000–U+3FFFD, or U+40000–U+4FFFD
identifier-head → U+50000–U+5FFFD, U+60000–U+6FFFD, U+70000–U+7FFFD, or U+80000–U+8FFFD
identifier-head → U+90000–U+9FFFD, U+A0000–U+AFFFD, U+B0000–U+BFFFD, or U+C0000–U+CFFFD
identifier-head → U+D0000–U+DFFFD or U+E0000–U+EFFFD
identifier-character → Digit 0 through 9
identifier-character → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–U+FE2F
identifier-character → *identifier-head*
identifier-characters → *identifier-character* *identifier-characters* *opt*
implicit-parameter-name → **$** *decimal-digits*
property-wrapper-projection → **$** *identifier-characters*

Converted by Evan at Apps Dissected -www.appsdissected.com

Keywords and Punctuation

The following keywords are reserved and can't be used as identifiers, unless they're escaped with backticks, as described above in Identifiers. Keywords other than `inout`, `var`, and `let` can be used as parameter names in a function declaration or function call without being escaped with backticks. When a member has the same name as a keyword, references to that member don't need to be escaped with backticks, except when there's ambiguity between referring to the member and using the keyword—for example, `self`, `Type`, and `Protocol` have special meaning in an explicit member expression, so they must be escaped with backticks in that context.

- Keywords used in declarations: `associatedtype`, `class`, `deinit`, `enum`, `extension`, `fileprivate`, `func`, `import`, `init`, `inout`, `internal`, `let`, `open`, `operator`, `private`, `precedencegroup`, `protocol`, `public`, `rethrows`, `static`, `struct`, `subscript`, `typealias`, and `var`.

- Keywords used in statements: `break`, `case`, `catch`, `continue`, `default`, `defer`, `do`, `else`, `fallthrough`, `for`, `guard`, `if`, `in`, `repeat`, `return`, `throw`, `switch`, `where`, and `while`.

- Keywords used in expressions and types: `Any`, `as`, `catch`, `false`, `is`, `nil`, `rethrows`, `self`, `Self`, `super`, `throw`, `throws`, `true`, and `try`.

- Keywords used in patterns: `_`.

- Keywords that begin with a number sign (`#`): `#available`, `#colorLiteral`, `#column`, `#dsohandle`, `#elseif`, `#else`, `#endif`, `#error`, `#fileID`, `#fileLiteral`, `#filePath`, `#file`, `#function`, `#if`, `#imageLiteral`, `#keyPath`, `#line`, `#selector`, `#sourceLocation`, and `#warning`.

- Keywords reserved in particular contexts: `associativity`, `convenience`, `didSet`, `dynamic`, `final`, `get`, `indirect`, `infix`, `lazy`, `left`, `mutating`, `none`, `nonmutating`, `optional`, `override`, `postfix`,

`precedence`, `prefix`, `Protocol`, `required`, `right`, `set`, `some`, `Type`, `unowned`, `weak`, and `willSet`. Outside the context in which they appear in the grammar, they can be used as identifiers.

The following tokens are reserved as punctuation and can't be used as custom operators: `(`, `)`, `{`, `}`, `[`, `]`, `.`, `,`, `:`, `;`, `=`, `@`, `#`, `&` (as a prefix operator), `->`, `` ` ``, `?`, and `!` (as a postfix operator).

Literals

A *literal* is the source code representation of a value of a type, such as a number or string.

The following are examples of literals:

```
42                // Integer literal
3.14159           // Floating-point literal
"Hello, world!"   // String literal
true              // Boolean literal
```

A literal doesn't have a type on its own. Instead, a literal is parsed as having infinite precision and Swift's type inference attempts to infer a type for the literal. For example, in the declaration `let x: Int8 = 42`, Swift uses the explicit type annotation (`: Int8`) to infer that the type of the integer literal `42` is `Int8`. If there isn't suitable type information available, Swift infers that the literal's type is one of the default literal types defined in the Swift standard library. The default types are `Int` for integer literals, `Double` for floating-point literals, `String` for string literals, and `Bool` for Boolean literals. For example, in the declaration `let str = "Hello, world"`, the default inferred type of the string literal `"Hello, world"` is `String`.

When specifying the type annotation for a literal value, the annotation's type must be a type that can be instantiated from that literal value. That is, the type must conform to one of the following Swift standard library protocols: `ExpressibleByIntegerLiteral` for integer literals, `ExpressibleByFloatLiteral` for floating-point literals, `ExpressibleByStringLiteral` for string literals, `ExpressibleByBooleanLiteral` for Boolean literals, `ExpressibleByUnicodeScalarLiteral` for string literals that contain only a single Unicode scalar, and `ExpressibleByExtendedGraphemeClusterLiteral` for string literals that contain only a single extended grapheme cluster. For example, `Int8` conforms to the `ExpressibleByIntegerLiteral` protocol, and therefore it can be used in the type annotation for the integer literal `42` in the declaration `let x: Int8 = 42`.

GRAMMAR OF A LITERAL

literal → *numeric-literal* | *string-literal* | *boolean-literal* | *nil-literal*
numeric-literal → **-**$_{opt}$ *integer-literal* | **-**$_{opt}$ *floating-point-literal*
boolean-literal → **true** | **false**
nil-literal → **nil**

Integer Literals

Integer literals represent integer values of unspecified precision. By default, integer literals are expressed in decimal; you can specify an alternate base using a prefix. Binary literals begin with `0b`, octal literals begin with `0o`, and hexadecimal literals begin with `0x`.

Decimal literals contain the digits `0` through `9`. Binary literals contain `0` and `1`, octal literals contain `0` through `7`, and hexadecimal literals contain `0` through `9` as well as `A` through `F` in upper- or lowercase.

Negative integers literals are expressed by prepending a minus sign (`-`) to an integer literal, as in `-42`.

Underscores (_) are allowed between digits for readability, but they're ignored and therefore don't affect the value of the literal. Integer literals can begin with leading zeros (`0`), but they're likewise ignored and don't affect the base or value of the literal.

Unless otherwise specified, the default inferred type of an integer literal is the Swift standard library type `Int`. The Swift standard library also defines types for various sizes of signed and unsigned integers, as described in Integers.

GRAMMAR OF AN INTEGER LITERAL

integer-literal → *binary-literal*
integer-literal → *octal-literal*
integer-literal → *decimal-literal*
integer-literal → *hexadecimal-literal*
binary-literal → **0b** *binary-digit* *binary-literal-characters* *opt*
binary-digit → Digit 0 or 1
binary-literal-character → *binary-digit* | _
binary-literal-characters → *binary-literal-character* *binary-literal-characters* *opt*
octal-literal → **0o** *octal-digit* *octal-literal-characters* *opt*
octal-digit → Digit 0 through 7
octal-literal-character → *octal-digit* | _
octal-literal-characters → *octal-literal-character* *octal-literal-characters* *opt*
decimal-literal → *decimal-digit* *decimal-literal-characters* *opt*
decimal-digit → Digit 0 through 9
decimal-digits → *decimal-digit* *decimal-digits* *opt*
decimal-literal-character → *decimal-digit* | _
decimal-literal-characters → *decimal-literal-character* *decimal-literal-characters* *opt*
hexadecimal-literal → **0x** *hexadecimal-digit* *hexadecimal-literal-characters* *opt*
hexadecimal-digit → Digit 0 through 9, a through f, or A through F
hexadecimal-literal-character → *hexadecimal-digit* | _
hexadecimal-literal-characters → *hexadecimal-literal-character* *hexadecimal-literal-characters* *opt*

Floating-Point Literals

Floating-point literals represent floating-point values of unspecified precision.

By default, floating-point literals are expressed in decimal (with no prefix), but they can also be expressed in hexadecimal (with a `0x` prefix).

Decimal floating-point literals consist of a sequence of decimal digits followed by either a decimal fraction, a decimal exponent, or both. The decimal fraction consists of a decimal point (`.`) followed by a sequence of decimal digits. The exponent consists of an upper- or lowercase `e` prefix followed by a sequence of decimal digits that indicates what power of 10 the value preceding the `e` is multiplied by. For example, `1.25e2` represents 1.25×10^2, which evaluates to `125.0`. Similarly, `1.25e-2` represents 1.25×10^{-2}, which evaluates to `0.0125`.

Hexadecimal floating-point literals consist of a `0x` prefix, followed by an optional hexadecimal fraction, followed by a hexadecimal exponent. The hexadecimal fraction consists of a decimal point followed by a sequence of hexadecimal digits. The exponent consists of an upper- or lowercase `p` prefix followed by a sequence of decimal digits that indicates what power of 2 the value preceding the `p` is multiplied by. For example, `0xFp2` represents 15×2^2, which evaluates to `60`. Similarly, `0xFp-2` represents 15×2^{-2}, which evaluates to `3.75`.

Negative floating-point literals are expressed by prepending a minus sign (–) to a floating-point literal, as in `-42.5`.

Underscores (_) are allowed between digits for readability, but they're ignored and therefore don't affect the value of the literal. Floating-point literals can begin with leading zeros (`0`), but they're likewise ignored and don't affect the base or value of the literal.

Unless otherwise specified, the default inferred type of a floating-point literal is the Swift standard library type `Double`, which represents a 64-bit floating-point number. The Swift standard library also defines a `Float` type, which represents a 32-bit floating-point number.

GRAMMAR OF A FLOATING-POINT LITERAL

floating-point-literal → *decimal-literal* *decimal-fraction* *opt* *decimal-exponent* *opt*
floating-point-literal → *hexadecimal-literal* *hexadecimal-fraction* *opt* *hexadecimal-exponent*
decimal-fraction → **.** *decimal-literal*
decimal-exponent → *floating-point-e* *sign* *opt* *decimal-literal*
hexadecimal-fraction → **.** *hexadecimal-digit* *hexadecimal-literal-characters* *opt*
hexadecimal-exponent → *floating-point-p* *sign* *opt* *decimal-literal*
floating-point-e → **e** | **E**
floating-point-p → **p** | **P**
sign → **+** | **-**

String Literals

A string literal is a sequence of characters surrounded by quotation marks. A single-line string literal is surrounded by double quotation marks and has the following form:

```
"characters"
```

String literals can't contain an unescaped double quotation mark ("), an unescaped backslash (\), a carriage return, or a line feed.

A multiline string literal is surrounded by three double quotation marks and has the following form:

```
"""
characters
"""
```

Unlike a single-line string literal, a multiline string literal can contain unescaped double quotation marks ("), carriage returns, and line feeds. It can't contain three unescaped double quotation marks next to each other.

The line break after the `"""` that begins the multiline string literal isn't part of the string. The line break before the `"""` that ends the literal is also not part of the string. To make a multiline string literal that begins or ends with a line feed, write a blank line as its first or last line.

A multiline string literal can be indented using any combination of spaces and tabs; this indentation isn't included in the string. The `"""` that ends the literal determines the indentation: Every nonblank line in the literal must begin with exactly the same indentation that appears before the closing `"""`; there's no conversion between tabs and spaces. You can include additional spaces and tabs after that indentation; those spaces and tabs appear in the string.

Line breaks in a multiline string literal are normalized to use the line feed character. Even if your source file has a mix of carriage returns and line feeds, all of the line breaks in the string will be the same.

In a multiline string literal, writing a backslash (`\`) at the end of a line omits that line break from the string. Any whitespace between the backslash and the line break is also omitted. You can use this syntax to hard wrap a multiline string literal in your source code, without changing the value of the resulting string.

Special characters can be included in string literals of both the single-line and multiline forms using the following escape sequences:

- Null character (`\0`)
- Backslash (`\\`)
- Horizontal tab (`\t`)
- Line feed (`\n`)
- Carriage return (`\r`)
- Double quotation mark (`\"`)

- Single quotation mark (`\'`)
- Unicode scalar (`\u{n}`), where *n* is a hexadecimal number that has one to eight digits

The value of an expression can be inserted into a string literal by placing the expression in parentheses after a backslash (`\`). The interpolated expression can contain a string literal, but can't contain an unescaped backslash, a carriage return, or a line feed.

For example, all of the following string literals have the same value:

```
"1 2 3"
"1 2 \("3")"
"1 2 \(3)"
"1 2 \(1 + 2)"
let x = 3; "1 2 \(x)"
```

A string delimited by extended delimiters is a sequence of characters surrounded by quotation marks and a balanced set of one or more number signs (#). A string delimited by extended delimiters has the following forms:

```
#"characters"#

#"""
characters
"""#
```

Special characters in a string delimited by extended delimiters appear in the resulting string as normal characters rather than as special characters. You can use extended delimiters to create strings with characters that would

ordinarily have a special effect such as generating a string interpolation, starting an escape sequence, or terminating the string.

The following example shows a string literal and a string delimited by extended delimiters that create equivalent string values:

```
let string = #"\(x) \ " \u{2603}"#
let escaped = "\\(x) \\ \" \\u{2603}"
print(string)
// Prints "\(x) \ " \u{2603}"
print(string == escaped)
// Prints "true"
```

If you use more than one number sign to form a string delimited by extended delimiters, don't place whitespace in between the number signs:

```
print(###"Line 1\###nLine 2"###) // OK
print(# # #"Line 1\# # #nLine 2"# # #) // Error
```

Multiline string literals that you create using extended delimiters have the same indentation requirements as regular multiline string literals.

The default inferred type of a string literal is `String`. For more information about the `String` type, see Strings and Characters and `String`.

String literals that are concatenated by the `+` operator are concatenated at compile time. For example, the values of `textA` and `textB` in the example below are identical—no runtime concatenation is performed.

```
let textA = "Hello " + "world"
let textB = "Hello world"
```

GRAMMAR OF A STRING LITERAL

string-literal → *static-string-literal* | *interpolated-string-literal*
string-literal-opening-delimiter → *extended-string-literal-delimiter* *opt* **"**
string-literal-closing-delimiter → **"** *extended-string-literal-delimiter* *opt*
static-string-literal → *string-literal-opening-delimiter* *quoted-text* *opt* *string-literal-closing-delimiter*
static-string-literal → *multiline-string-literal-opening-delimiter* *multiline-quoted-text* *opt* *multiline-string-literal-closing-delimiter*
multiline-string-literal-opening-delimiter → *extended-string-literal-delimiter* *opt* **"""**
multiline-string-literal-closing-delimiter → **"""** *extended-string-literal-delimiter* *opt*
extended-string-literal-delimiter → **#** *extended-string-literal-delimiter* *opt*
quoted-text → *quoted-text-item* *quoted-text* *opt*
quoted-text-item → *escaped-character*
quoted-text-item → Any Unicode scalar value except **"**, ****, U+000A, or U+000D
multiline-quoted-text → *multiline-quoted-text-item* *multiline-quoted-text* *opt*
multiline-quoted-text-item → *escaped-character*
multiline-quoted-text-item → Any Unicode scalar value except ****
multiline-quoted-text-item → *escaped-newline*
interpolated-string-literal → *string-literal-opening-delimiter* *interpolated-text* *opt* *string-literal-closing-delimiter*
interpolated-string-literal → *multiline-string-literal-opening-delimiter* *multiline-interpolated-text* *opt* *multiline-string-literal-closing-delimiter*
interpolated-text → *interpolated-text-item* *interpolated-text* *opt*
interpolated-text-item → **\(** *expression* **)** | *quoted-text-item*
multiline-interpolated-text → *multiline-interpolated-text-item* *multiline-interpolated-text* *opt*
multiline-interpolated-text-item → **\(** *expression* **)** | *multiline-quoted-text-item*
escape-sequence → **** *extended-string-literal-delimiter*
escaped-character → *escape-sequence* **0** | *escape-sequence* **** | *escape-sequence* **t** | *escape-sequence* **n** | *escape-sequence* **r** | *escape-sequence* **"** | *escape-sequence* **'**
escaped-character → *escape-sequence* **u** **{** *unicode-scalar-digits* **}**
unicode-scalar-digits → Between one and eight hexadecimal digits
escaped-newline → *escape-sequence* *inline-spaces* *opt* *line-break*

Converted by Evan at Apps Dissected -www.appsdissected.com

Operators

The Swift standard library defines a number of operators for your use, many of which are discussed in Basic Operators and Advanced Operators. The present section describes which characters can be used to define custom operators.

Custom operators can begin with one of the ASCII characters `/`, `=`, `-`, `+`, `!`, `*`, `%`, `<`, `>`, `&`, `|`, `^`, `?`, or `~`, or one of the Unicode characters defined in the grammar below (which include characters from the *Mathematical Operators*, *Miscellaneous Symbols*, and *Dingbats* Unicode blocks, among others). After the first character, combining Unicode characters are also allowed.

You can also define custom operators that begin with a dot (`.`). These operators can contain additional dots. For example, `.+.` is treated as a single operator. If an operator doesn't begin with a dot, it can't contain a dot elsewhere. For example, `+.+` is treated as the `+` operator followed by the `.+` operator.

Although you can define custom operators that contain a question mark (`?`), they can't consist of a single question mark character only. Additionally, although operators can contain an exclamation point (`!`), postfix operators can't begin with either a question mark or an exclamation point.

> NOTE
>
> The tokens `=`, `->`, `//`, `/*`, `*/`, `.`, the prefix operators `<`, `&`, and `?`, the infix operator `?`, and the postfix operators `>`, `!`, and `?` are reserved. These tokens can't be overloaded, nor can they be used as custom operators.

The whitespace around an operator is used to determine whether an operator is used as a prefix operator, a postfix operator, or a binary operator. This behavior is summarized in the following rules:

- If an operator has whitespace around both sides or around neither side, it's treated as a binary operator. As an example, the `+++` operator in

`a+++b` and `a +++ b` is treated as a binary operator.

- If an operator has whitespace on the left side only, it's treated as a prefix unary operator. As an example, the `+++` operator in `a +++b` is treated as a prefix unary operator.

- If an operator has whitespace on the right side only, it's treated as a postfix unary operator. As an example, the `+++` operator in `a+++ b` is treated as a postfix unary operator.

- If an operator has no whitespace on the left but is followed immediately by a dot (`.`), it's treated as a postfix unary operator. As an example, the `+++` operator in `a+++.b` is treated as a postfix unary operator (`a+++ .b` rather than `a +++ .b`).

For the purposes of these rules, the characters `(`, `[`, and `{` before an operator, the characters `)`, `]`, and `}` after an operator, and the characters `,`, `;`, and `:` are also considered whitespace.

There's one caveat to the rules above. If the `!` or `?` predefined operator has no whitespace on the left, it's treated as a postfix operator, regardless of whether it has whitespace on the right. To use the `?` as the optional-chaining operator, it must not have whitespace on the left. To use it in the ternary conditional (`? :`) operator, it must have whitespace around both sides.

In certain constructs, operators with a leading `<` or `>` may be split into two or more tokens. The remainder is treated the same way and may be split again. As a result, you don't need to add whitespace to disambiguate between the closing `>` characters in constructs like `Dictionary<String, Array<Int>>`. In this example, the closing `>` characters aren't treated as a single token that may then be misinterpreted as a bit shift `>>` operator.

To learn how to define new, custom operators, see Custom Operators and Operator Declaration. To learn how to overload existing operators, see Operator Methods.

GRAMMAR OF OPERATORS

operator → *operator-head* *operator-characters* *opt*
operator → *dot-operator-head* *dot-operator-characters*
operator-head → **/** | **=** | **-** | **+** | **!** | ***** | **%** | **<** | **>** | **&** | **|** | **^** | **~** | **?**
operator-head → U+00A1–U+00A7
operator-head → U+00A9 or U+00AB
operator-head → U+00AC or U+00AE
operator-head → U+00B0–U+00B1
operator-head → U+00B6, U+00BB, U+00BF, U+00D7, or U+00F7
operator-head → U+2016–U+2017
operator-head → U+2020–U+2027
operator-head → U+2030–U+203E
operator-head → U+2041–U+2053
operator-head → U+2055–U+205E
operator-head → U+2190–U+23FF
operator-head → U+2500–U+2775
operator-head → U+2794–U+2BFF
operator-head → U+2E00–U+2E7F
operator-head → U+3001–U+3003
operator-head → U+3008–U+3020
operator-head → U+3030
operator-character → *operator-head*
operator-character → U+0300–U+036F
operator-character → U+1DC0–U+1DFF
operator-character → U+20D0–U+20FF
operator-character → U+FE00–U+FE0F
operator-character → U+FE20–U+FE2F
operator-character → U+E0100–U+E01EF
operator-characters → *operator-character* *operator-characters* *opt*
dot-operator-head → **.**
dot-operator-character → **.** | *operator-character*
dot-operator-characters → *dot-operator-character* *dot-operator-characters* *opt*
binary-operator → *operator*
prefix-operator → *operator*
postfix-operator → *operator*

Converted by Evan at Apps Dissected -www.appsdissected.com

Types

In Swift, there are two kinds of types: named types and compound types. A *named type* is a type that can be given a particular name when it's defined. Named types include classes, structures, enumerations, and protocols. For example, instances of a user-defined class named `MyClass` have the type `MyClass`. In addition to user-defined named types, the Swift standard library defines many commonly used named types, including those that represent arrays, dictionaries, and optional values.

Data types that are normally considered basic or primitive in other languages—such as types that represent numbers, characters, and strings—are actually named types, defined and implemented in the Swift standard library using structures. Because they're named types, you can extend their behavior to suit the needs of your program, using an extension declaration, discussed in Extensions and Extension Declaration.

A *compound type* is a type without a name, defined in the Swift language itself. There are two compound types: function types and tuple types. A compound type may contain named types and other compound types. For example, the tuple type `(Int, (Int, Int))` contains two elements: The first is the named type `Int`, and the second is another compound type `(Int, Int)`.

You can put parentheses around a named type or a compound type. However, adding parentheses around a type doesn't have any effect. For example, `(Int)` is equivalent to `Int`.

This chapter discusses the types defined in the Swift language itself and describes the type inference behavior of Swift.

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A TYPE

type → *function-type*
type → *array-type*
type → *dictionary-type*
type → *type-identifier*
type → *tuple-type*
type → *optional-type*
type → *implicitly-unwrapped-optional-type*
type → *protocol-composition-type*
type → *opaque-type*
type → *metatype-type*
type → *any-type*
type → *self-type*
type → **(** *type* **)**

Type Annotation

A *type annotation* explicitly specifies the type of a variable or expression. Type annotations begin with a colon (`:`) and end with a type, as the following examples show:

```
let someTuple: (Double, Double) = (3.14159, 2.71828)
func someFunction(a: Int) { /* ... */ }
```

In the first example, the expression `someTuple` is specified to have the tuple type `(Double, Double)`. In the second example, the parameter `a` to the function `someFunction` is specified to have the type `Int`.

Type annotations can contain an optional list of type attributes before the type.

GRAMMAR OF A TYPE ANNOTATION

type-annotation → **:** *attributes* $_{opt}$ **inout**$_{opt}$ *type*

Type Identifier

A *type identifier* refers to either a named type or a type alias of a named or compound type.

Most of the time, a type identifier directly refers to a named type with the same name as the identifier. For example, `Int` is a type identifier that directly refers to the named type `Int`, and the type identifier `Dictionary<String, Int>` directly refers to the named type `Dictionary<String, Int>`.

There are two cases in which a type identifier doesn't refer to a type with the same name. In the first case, a type identifier refers to a type alias of a named or compound type. For instance, in the example below, the use of `Point` in the type annotation refers to the tuple type `(Int, Int)`.

```
typealias Point = (Int, Int)
let origin: Point = (0, 0)
```

In the second case, a type identifier uses dot (`.`) syntax to refer to named types declared in other modules or nested within other types. For example, the type identifier in the following code references the named type `MyType` that's declared in the `ExampleModule` module.

```
var someValue: ExampleModule.MyType
```

GRAMMAR OF A TYPE IDENTIFIER

type-identifier → *type-name* *generic-argument-clause* *opt* | *type-name* *generic-argument-clause* *opt* . *type-identifier*

type-name → *identifier*

Tuple Type

A *tuple type* is a comma-separated list of types, enclosed in parentheses.

You can use a tuple type as the return type of a function to enable the function to return a single tuple containing multiple values. You can also name the elements of a tuple type and use those names to refer to the values of the individual elements. An element name consists of an identifier followed immediately by a colon (:). For an example that demonstrates both of these features, see Functions with Multiple Return Values.

When an element of a tuple type has a name, that name is part of the type.

```
var someTuple = (top: 10, bottom: 12)  // someTuple is of type (top: Int, bottom: Int)
someTuple = (top: 4, bottom: 42) // OK: names match
someTuple = (9, 99)              // OK: names are inferred
someTuple = (left: 5, right: 5)  // Error: names don't match
```

All tuple types contain two or more types, except for `Void` which is a type alias for the empty tuple type, `()`.

GRAMMAR OF A TUPLE TYPE

tuple-type → **(** **)** | **(** *tuple-type-element* **,** *tuple-type-element-list* **)**
tuple-type-element-list → *tuple-type-element* | *tuple-type-element* **,** *tuple-type-element-list*
tuple-type-element → *element-name* *type-annotation* | *type*
element-name → *identifier*

Function Type

A *function type* represents the type of a function, method, or closure and consists of a parameter and return type separated by an arrow (`->`):

```
(parameter type) -> return type
```

The *parameter type* is comma-separated list of types. Because the *return type* can be a tuple type, function types support functions and methods that return multiple values.

A parameter of the function type `() -> T` (where `T` is any type) can apply the `autoclosure` attribute to implicitly create a closure at its call sites. This provides a syntactically convenient way to defer the evaluation of an expression without needing to write an explicit closure when you call the function. For an example of an autoclosure function type parameter, see Autoclosures.

A function type can have variadic parameters in its *parameter type*. Syntactically, a variadic parameter consists of a base type name followed immediately by three dots (`...`), as in `Int...`. A variadic parameter is treated as an array that contains elements of the base type name. For instance, the variadic parameter `Int...` is treated as `[Int]`. For an example that uses a variadic parameter, see Variadic Parameters.

To specify an in-out parameter, prefix the parameter type with the `inout` keyword. You can't mark a variadic parameter or a return type with the `inout` keyword. In-out parameters are discussed in In-Out Parameters.

If a function type has only one parameter and that parameter's type is a tuple type, then the tuple type must be parenthesized when writing the function's type. For example, `((Int, Int)) -> Void` is the type of a function that takes a single parameter of the tuple type `(Int, Int)` and doesn't return any value. In contrast, without parentheses, `(Int, Int) -> Void` is the type of a function that takes two `Int` parameters and doesn't return any value. Likewise, because `Void` is a type alias for `()`, the function

type `(Void) -> Void` is the same as `(()) -> ()`—a function that takes a single argument that's an empty tuple. These types aren't the same as `() -> ()`—a function that takes no arguments.

Argument names in functions and methods aren't part of the corresponding function type. For example:

```
func someFunction(left: Int, right: Int) {}
func anotherFunction(left: Int, right: Int) {}
func functionWithDifferentLabels(top: Int, bottom: Int) {}

var f = someFunction // The type of f is (Int, Int) -> Void, not (left: Int, right: Int) -> Void.
f = anotherFunction              // OK
f = functionWithDifferentLabels  // OK

func functionWithDifferentArgumentTypes(left: Int, right: String) {}
f = functionWithDifferentArgumentTypes     // Error

func functionWithDifferentNumberOfArguments(left: Int, right: Int, top: Int) {}
f = functionWithDifferentNumberOfArguments // Error
```

Because argument labels aren't part of a function's type, you omit them when writing a function type.

```
var operation: (lhs: Int, rhs: Int) -> Int     // Error
var operation: (_ lhs: Int, _ rhs: Int) -> Int // OK
var operation: (Int, Int) -> Int               // OK
```

If a function type includes more than a single arrow (`->`), the function types are grouped from right to left. For example, the function type `(Int) -> (Int) -> Int` is understood as `(Int) -> ((Int) -> Int)`—that is, a function that takes an `Int` and returns another function that takes and returns an `Int`.

Function types that can throw or rethrow an error must be marked with the `throws` keyword. The `throws` keyword is part of a function's type, and nonthrowing functions are subtypes of throwing functions. As a result, you can use a nonthrowing function in the same places as a throwing one. Throwing and rethrowing functions are described in [Throwing Functions and Methods] and [Rethrowing Functions and Methods].

Restrictions for Nonescaping Closures

A parameter that's a nonescaping function can't be stored in a property, variable, or constant of type `Any`, because that might allow the value to escape.

A parameter that's a nonescaping function can't be passed as an argument to another nonescaping function parameter. This restriction helps Swift perform more of its checks for conflicting access to memory at compile time instead of at runtime. For example:

```
let external: (() -> Void) -> Void = { _ in () }
func takesTwoFunctions(first: (() -> Void) -> Void,
     second: (() -> Void) -> Void) {
    first { first {} }       // Error
    second { second {}  }    // Error

    first { second {} }      // Error
    second { first {} }      // Error

    first { external {} }    // OK
    external { first {} }    // OK
}
```

In the code above, both of the parameters to `takesTwoFunctions(first:second:)` are functions. Neither parameter is marked `@escaping`, so they're both nonescaping as a result.

The four function calls marked "Error" in the example above cause compiler errors. Because the `first` and `second` parameters are nonescaping functions, they can't be passed as arguments to another nonescaping function parameter. In contrast, the two function calls marked "OK" don't cause a compiler error. These function calls don't violate the restriction because `external` isn't one of the parameters of `takesTwoFunctions(first:second:)`.

If you need to avoid this restriction, mark one of the parameters as escaping, or temporarily convert one of the nonescaping function parameters to an escaping function by using the `withoutActuallyEscaping(_:do:)` function. For information about avoiding conflicting access to memory, see Memory Safety.

GRAMMAR OF A FUNCTION TYPE

function-type → *attributes* *opt* *function-type-argument-clause* **throws** *opt* **->** *type*
function-type-argument-clause → **(** **)**
function-type-argument-clause → **(** *function-type-argument-list* **...** *opt* **)**
function-type-argument-list → *function-type-argument* | *function-type-argument* **,** *function-type-argument-list*
function-type-argument → *attributes* *opt* **inout** *opt* *type* | *argument-label* *type-annotation*
argument-label → *identifier*

Array Type

The Swift language provides the following syntactic sugar for the Swift standard library `Array<Element>` type:

```
[type]
```

In other words, the following two declarations are equivalent:

```
let someArray: Array<String> = ["Alex", "Brian", "Dave"]
let someArray: [String] = ["Alex", "Brian", "Dave"]
```

In both cases, the constant `someArray` is declared as an array of strings. The elements of an array can be accessed through subscripting by specifying a valid index value in square brackets: `someArray[0]` refers to the element at index 0, `"Alex"`.

You can create multidimensional arrays by nesting pairs of square brackets, where the name of the base type of the elements is contained in the

innermost pair of square brackets. For example, you can create a three-dimensional array of integers using three sets of square brackets:

```
var array3D: [[[Int]]] = [[[1, 2], [3, 4]], [[5, 6],
    [7, 8]]]
```

When accessing the elements in a multidimensional array, the left-most subscript index refers to the element at that index in the outermost array. The next subscript index to the right refers to the element at that index in the array that's nested one level in. And so on. This means that in the example above, `array3D[0]` refers to `[[1, 2], [3, 4]]`, `array3D[0][1]` refers to `[3, 4]`, and `array3D[0][1][1]` refers to the value 4.

For a detailed discussion of the Swift standard library `Array` type, see Arrays.

GRAMMAR OF AN ARRAY TYPE

array-type → **[** *type* **]**

Dictionary Type

The Swift language provides the following syntactic sugar for the Swift standard library `Dictionary<Key, Value>` type:

```
[key type: value type]
```

In other words, the following two declarations are equivalent:

```
1 let someDictionary: [String: Int] = ["Alex": 31,
      "Paul": 39]
2 let someDictionary: Dictionary<String, Int> =
      ["Alex": 31, "Paul": 39]
```

In both cases, the constant `someDictionary` is declared as a dictionary with strings as keys and integers as values.

The values of a dictionary can be accessed through subscripting by specifying the corresponding key in square brackets: `someDictionary["Alex"]` refers to the value associated with the key `"Alex"`. The subscript returns an optional value of the dictionary's value type. If the specified key isn't contained in the dictionary, the subscript returns `nil`.

The key type of a dictionary must conform to the Swift standard library `Hashable` protocol.

For a detailed discussion of the Swift standard library `Dictionary` type, see Dictionaries.

GRAMMAR OF A DICTIONARY TYPE

dictionary-type → **[** *type* **:** *type* **]**

Optional Type

The Swift language defines the postfix `?` as syntactic sugar for the named type `Optional<Wrapped>`, which is defined in the Swift standard library. In other words, the following two declarations are equivalent:

```
var optionalInteger: Int?
var optionalInteger: Optional<Int>
```

In both cases, the variable `optionalInteger` is declared to have the type of an optional integer. Note that no whitespace may appear between the type and the `?`.

The type `Optional<Wrapped>` is an enumeration with two cases, `none` and `some(Wrapped)`, which are used to represent values that may or may not be present. Any type can be explicitly declared to be (or implicitly converted to) an optional type. If you don't provide an initial value when you declare an optional variable or property, its value automatically defaults to `nil`.

If an instance of an optional type contains a value, you can access that value using the postfix operator `!`, as shown below:

```
optionalInteger = 42
optionalInteger! // 42
```

Using the `!` operator to unwrap an optional that has a value of `nil` results in a runtime error.

You can also use optional chaining and optional binding to conditionally perform an operation on an optional expression. If the value is `nil`, no operation is performed and therefore no runtime error is produced.

For more information and to see examples that show how to use optional types, see Optionals.

GRAMMAR OF AN OPTIONAL TYPE

optional-type → *type* **?**

Implicitly Unwrapped Optional Type

The Swift language defines the postfix `!` as syntactic sugar for the named type `Optional<Wrapped>`, which is defined in the Swift standard library, with the additional behavior that it's automatically unwrapped when it's accessed. If you try to use an implicitly unwrapped optional that has a value of `nil`, you'll get a runtime error. With the exception of the implicit unwrapping behavior, the following two declarations are equivalent:

```
var implicitlyUnwrappedString: String!
var explicitlyUnwrappedString: Optional<String>
```

Note that no whitespace may appear between the type and the `!`.

Because implicit unwrapping changes the meaning of the declaration that contains that type, optional types that are nested inside a tuple type or a generic type—such as the element types of a dictionary or array—can't be marked as implicitly unwrapped. For example:

```
let tupleOfImplicitlyUnwrappedElements: (Int!, Int!)
     // Error
let implicitlyUnwrappedTuple: (Int, Int)!
     // OK

let arrayOfImplicitlyUnwrappedElements: [Int!]
     // Error
let implicitlyUnwrappedArray: [Int]!
     // OK
```

Because implicitly unwrapped optionals have the same `Optional<Wrapped>` type as optional values, you can use implicitly

unwrapped optionals in all the same places in your code that you can use optionals. For example, you can assign values of implicitly unwrapped optionals to variables, constants, and properties of optionals, and vice versa.

As with optionals, if you don't provide an initial value when you declare an implicitly unwrapped optional variable or property, its value automatically defaults to `nil`.

Use optional chaining to conditionally perform an operation on an implicitly unwrapped optional expression. If the value is `nil`, no operation is performed and therefore no runtime error is produced.

For more information about implicitly unwrapped optional types, see Implicitly Unwrapped Optionals.

GRAMMAR OF AN IMPLICITLY UNWRAPPED OPTIONAL TYPE

implicitly-unwrapped-optional-type → *type* !

Protocol Composition Type

A *protocol composition type* defines a type that conforms to each protocol in a list of specified protocols, or a type that's a subclass of a given class and conforms to each protocol in a list of specified protocols. Protocol composition types may be used only when specifying a type in type annotations, in generic parameter clauses, and in generic `where` clauses.

Protocol composition types have the following form:

```
Protocol 1 & Protocol 2
```

A protocol composition type allows you to specify a value whose type conforms to the requirements of multiple protocols without explicitly defining a new, named protocol that inherits from each protocol you want

the type to conform to. For example, you can use the protocol composition type `ProtocolA & ProtocolB & ProtocolC` instead of declaring a new protocol that inherits from `ProtocolA`, `ProtocolB`, and `ProtocolC`. Likewise, you can use `SuperClass & ProtocolA` instead of declaring a new protocol that's a subclass of `SuperClass` and conforms to `ProtocolA`.

Each item in a protocol composition list is one of the following; the list can contain at most one class:

- The name of a class
- The name of a protocol
- A type alias whose underlying type is a protocol composition type, a protocol, or a class.

When a protocol composition type contains type aliases, it's possible for the same protocol to appear more than once in the definitions—duplicates are ignored. For example, the definition of `PQR` in the code below is equivalent to `P & Q & R`.

```
typealias PQ = P & Q
typealias PQR = PQ & Q & R
```

GRAMMAR OF A PROTOCOL COMPOSITION TYPE

protocol-composition-type → *type-identifier* **&** *protocol-composition-continuation*
protocol-composition-continuation → *type-identifier* | *protocol-composition-type*

Opaque Type

An *opaque type* defines a type that conforms to a protocol or protocol composition, without specifying the underlying concrete type.

Opaque types appear as the return type of a function or subscript, or the type of a property. Opaque types can't appear as part of a tuple type or a generic type, such as the element type of an array or the wrapped type of an optional.

Opaque types have the following form:

```
some constraint
```

The *constraint* is a class type, protocol type, protocol composition type, or `Any`. A value can be used as an instance of the opaque type only if it's an instance of a type that conforms to the listed protocol or protocol composition, or inherits from the listed class. Code that interacts with an opaque value can use the value only in ways that are part of the interface defined by the *constraint*.

Protocol declarations can't include opaque types. Classes can't use an opaque type as the return type of a nonfinal method.

A function that uses an opaque type as its return type must return values that share a single underlying type. The return type can include types that are part of the function's generic type parameters. For example, a function `someFunction<T>()` could return a value of type `T` or `Dictionary<String, T>`.

GRAMMAR OF AN OPAQUE TYPE

opaque-type → **some** *type*

Metatype Type

A *metatype type* refers to the type of any type, including class types, structure types, enumeration types, and protocol types.

The metatype of a class, structure, or enumeration type is the name of that type followed by `.Type`. The metatype of a protocol type—not the concrete type that conforms to the protocol at runtime—is the name of that protocol followed by `.Protocol`. For example, the metatype of the class type `SomeClass` is `SomeClass.Type` and the metatype of the protocol `SomeProtocol` is `SomeProtocol.Protocol`.

You can use the postfix `self` expression to access a type as a value. For example, `SomeClass.self` returns `SomeClass` itself, not an instance of `SomeClass`. And `SomeProtocol.self` returns `SomeProtocol` itself, not an instance of a type that conforms to `SomeProtocol` at runtime. You can call the `type(of:)` function with an instance of a type to access that instance's dynamic, runtime type as a value, as the following example shows:

```swift
class SomeBaseClass {
    class func printClassName() {
        print("SomeBaseClass")
    }
}
class SomeSubClass: SomeBaseClass {
    override class func printClassName() {
        print("SomeSubClass")
    }
}
let someInstance: SomeBaseClass = SomeSubClass()
// The compile-time type of someInstance is SomeBaseClass,
// and the runtime type of someInstance is SomeSubClass
type(of: someInstance).printClassName()
// Prints "SomeSubClass"
```

For more information, see `type(of:)` in the Swift standard library.

Use an initializer expression to construct an instance of a type from that type's metatype value. For class instances, the initializer that's called must be marked with the `required` keyword or the entire class marked with the `final` keyword.

```
class AnotherSubClass: SomeBaseClass {
    let string: String
    required init(string: String) {
        self.string = string
    }
    override class func printClassName() {
        print("AnotherSubClass")
    }
}
let metatype: AnotherSubClass.Type =
     AnotherSubClass.self
let anotherInstance = metatype.init(string: "some
     string")
```

GRAMMAR OF A METATYPE TYPE

metatype-type → *type* **.** **Type** | *type* **.** **Protocol**

Any Type

The `Any` type can contain values from all other types. `Any` can be used as the concrete type for an instance of any of the following types:

- A class, structure, or enumeration
- A metatype, such as `Int.self`
- A tuple with any types of components

- A closure or function type

```
let mixed: [Any] = ["one", 2, true, (4, 5.3), { () -
     > Int in return 6 }]
```

When you use `Any` as a concrete type for an instance, you need to cast the instance to a known type before you can access its properties or methods. Instances with a concrete type of `Any` maintain their original dynamic type and can be cast to that type using one of the type-cast operators—`as`, `as?`, or `as!`. For example, use `as?` to conditionally downcast the first object in a heterogeneous array to a `String` as follows:

```
if let first = mixed.first as? String {
    print("The first item, '\(first)', is a
     string.")
}
// Prints "The first item, 'one', is a string."
```

For more information about casting, see Type Casting.

The `AnyObject` protocol is similar to the `Any` type. All classes implicitly conform to `AnyObject`. Unlike `Any`, which is defined by the language, `AnyObject` is defined by the Swift standard library. For more information, see Class-Only Protocols and `AnyObject`.

GRAMMAR OF AN ANY TYPE

any-type → **Any**

Self Type

The `Self` type isn't a specific type, but rather lets you conveniently refer to the current type without repeating or knowing that type's name.

In a protocol declaration or a protocol member declaration, the `Self` type refers to the eventual type that conforms to the protocol.

In a structure, class, or enumeration declaration, the `Self` type refers to the type introduced by the declaration. Inside the declaration for a member of a type, the `Self` type refers to that type. In the members of a class declaration, `Self` can appear only as follows:

- As the return type of a method
- As the return type of a read-only subscript
- As the type of a read-only computed property
- In the body of a method

For example, the code below shows an instance method `f` whose return type is `Self`.

```swift
class Superclass {
    func f() -> Self { return self }
}
let x = Superclass()
print(type(of: x.f()))
// Prints "Superclass"

class Subclass: Superclass { }
let y = Subclass()
print(type(of: y.f()))
// Prints "Subclass"

let z: Superclass = Subclass()
print(type(of: z.f()))
// Prints "Subclass"
```

The last part of the example above shows that `Self` refers to the runtime type `Subclass` of the value of `z`, not the compile-time type `Superclass` of the variable itself.

Inside a nested type declaration, the `Self` type refers to the type introduced by the innermost type declaration.

The `Self` type refers to the same type as the `type(of:)` function in the Swift standard library. Writing `Self.someStaticMember` to access a member of the current type is the same as writing `type(of: self).someStaticMember`.

GRAMMAR OF A SELF TYPE

self-type → **Self**

Type Inheritance Clause

A *type inheritance clause* is used to specify which class a named type inherits from and which protocols a named type conforms to. A type inheritance clause begins with a colon (`:`), followed by a list of type identifiers.

Class types can inherit from a single superclass and conform to any number of protocols. When defining a class, the name of the superclass must appear first in the list of type identifiers, followed by any number of protocols the class must conform to. If the class doesn't inherit from another class, the list can begin with a protocol instead. For an extended discussion and several examples of class inheritance, see Inheritance.

Other named types can only inherit from or conform to a list of protocols. Protocol types can inherit from any number of other protocols. When a protocol type inherits from other protocols, the set of requirements from those other protocols are aggregated together, and any type that inherits from the current protocol must conform to all of those requirements.

A type inheritance clause in an enumeration definition can be either a list of protocols, or in the case of an enumeration that assigns raw values to its cases, a single, named type that specifies the type of those raw values. For an example of an enumeration definition that uses a type inheritance clause to specify the type of its raw values, see Raw Values.

GRAMMAR OF A TYPE INHERITANCE CLAUSE

type-inheritance-clause → **:** *type-inheritance-list*
type-inheritance-list → *type-identifier* | *type-identifier* **,** *type-inheritance-list*

Type Inference

Swift uses *type inference* extensively, allowing you to omit the type or part of the type of many variables and expressions in your code. For example, instead of writing `var x: Int = 0`, you can write `var x = 0`, omitting the type completely—the compiler correctly infers that `x` names a value of type `Int`. Similarly, you can omit part of a type when the full type can be inferred from context. For example, if you write `let dict: Dictionary = ["A": 1]`, the compiler infers that `dict` has the type `Dictionary<String, Int>`.

In both of the examples above, the type information is passed up from the leaves of the expression tree to its root. That is, the type of `x` in `var x: Int = 0` is inferred by first checking the type of `0` and then passing this type information up to the root (the variable `x`).

In Swift, type information can also flow in the opposite direction—from the root down to the leaves. In the following example, for instance, the explicit type annotation (`: Float`) on the constant `eFloat` causes the numeric literal `2.71828` to have an inferred type of `Float` instead of `Double`.

```
1  let e = 2.71828 // The type of e is inferred to be
       Double.
2  let eFloat: Float = 2.71828 // The type of eFloat is
       Float.
```

Type inference in Swift operates at the level of a single expression or statement. This means that all of the information needed to infer an omitted type or part of a type in an expression must be accessible from type-checking the expression or one of its subexpressions.

Expressions

In Swift, there are four kinds of expressions: prefix expressions, binary expressions, primary expressions, and postfix expressions. Evaluating an expression returns a value, causes a side effect, or both.

Prefix and binary expressions let you apply operators to smaller expressions. Primary expressions are conceptually the simplest kind of expression, and they provide a way to access values. Postfix expressions, like prefix and binary expressions, let you build up more complex expressions using postfixes such as function calls and member access. Each kind of expression is described in detail in the sections below.

GRAMMAR OF AN EXPRESSION

expression → *try-operator* opt *await-operator* opt *prefix-expression* *binary-expressions* opt
expression-list → *expression* | *expression* , *expression-list*

Prefix Expressions

Prefix expressions combine an optional prefix operator with an expression. Prefix operators take one argument, the expression that follows them.

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

For information about the operators provided by the Swift standard library, see Operator Declarations.

GRAMMAR OF A PREFIX EXPRESSION

prefix-expression → *prefix-operator* opt *postfix-expression*
prefix-expression → *in-out-expression*

In-Out Expression

An *in-out expression* marks a variable that's being passed as an in-out argument to a function call expression.

```
&expression
```

For more information about in-out parameters and to see an example, see [In-Out Parameters](#).

In-out expressions are also used when providing a non-pointer argument in a context where a pointer is needed, as described in [Implicit Conversion to a Pointer Type](#).

GRAMMAR OF AN IN-OUT EXPRESSION

in-out-expression → **&** *identifier*

Try Operator

A *try expression* consists of the `try` operator followed by an expression that can throw an error. It has the following form:

```
try expression
```

The value of a `try` expression is the value of the *expression*.

An *optional-try expression* consists of the `try?` operator followed by an expression that can throw an error. It has the following form:

```
try? expression
```

If the *expression* doesn't throw an error, the value of the optional-try expression is an optional containing the value of the *expression*. Otherwise, the value of the optional-try expression is `nil`.

A *forced-try expression* consists of the `try!` operator followed by an expression that can throw an error. It has the following form:

```
try! expression
```

The value of a forced-try expression is the value of the *expression*. If the *expression* throws an error, a runtime error is produced.

When the expression on the left-hand side of a binary operator is marked with `try`, `try?`, or `try!`, that operator applies to the whole binary expression. That said, you can use parentheses to be explicit about the scope of the operator's application.

```
// try applies to both function calls
sum = try someThrowingFunction() +
     anotherThrowingFunction()

// try applies to both function calls
sum = try (someThrowingFunction() +
     anotherThrowingFunction())

// Error: try applies only to the first function
     call
sum = (try someThrowingFunction()) +
     anotherThrowingFunction()
```

A `try` expression can't appear on the right-hand side of a binary operator, unless the binary operator is the assignment operator or the `try` expression is enclosed in parentheses.

If an expression includes both the `try` and `await` operator, the `try` operator must appear first.

For more information and to see examples of how to use `try`, `try?`, and `try!`, see Error Handling.

```
GRAMMAR OF A TRY EXPRESSION
try-operator → try | try ? | try !
```

Await Operator

An *await expression* consists of the `await` operator followed by an expression that uses the result of an asynchronous operation. It has the following form:

```
await expression
```

The value of an `await` expression is the value of the *expression*.

An expression marked with `await` is called a *potential suspension point*. Execution of an asynchronous function can be suspended at each expression that's marked with `await`. In addition, execution of concurrent code is never suspended at any other point. This means code between potential suspension points can safely update state that requires temporarily breaking invariants, provided that it completes the update before the next potential suspension point.

An `await` expression can appear only within an asynchronous context, such as the trailing closure passed to the `async(priority:operation:)` function. It can't appear in the body of a `defer` statement, or in an autoclosure of synchronous function type.

When the expression on the left-hand side of a binary operator is marked with the `await` operator, that operator applies to the whole binary

expression. That said, you can use parentheses to be explicit about the scope of the operator's application.

```
// await applies to both function calls
sum = await someAsyncFunction() +
     anotherAsyncFunction()

// await applies to both function calls
sum = await (someAsyncFunction() +
     anotherAsyncFunction())

// Error: await applies only to the first function
     call
sum = (await someAsyncFunction()) +
     anotherAsyncFunction()
```

An `await` expression can't appear on the right-hand side of a binary operator, unless the binary operator is the assignment operator or the `await` expression is enclosed in parentheses.

If an expression includes both the `await` and `try` operator, the `try` operator must appear first.

GRAMMAR OF AN AWAIT EXPRESSION

await-operator → **`await`**

Binary Expressions

Binary expressions combine an infix binary operator with the expression that it takes as its left- and right-hand arguments. It has the following form:

`left-hand argument` `operator` `right-hand argument`

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

For information about the operators provided by the Swift standard library, see Operator Declarations.

> NOTE
>
> At parse time, an expression made up of binary operators is represented as a flat list. This list is transformed into a tree by applying operator precedence. For example, the expression `2 + 3 * 5` is initially understood as a flat list of five items, `2`, `+`, `3`, `*`, and `5`. This process transforms it into the tree (2 + (3 * 5)).

GRAMMAR OF A BINARY EXPRESSION

binary-expression → *binary-operator* *prefix-expression*
binary-expression → *assignment-operator* *try-operator* $_{opt}$ *prefix-expression*
binary-expression → *conditional-operator* *try-operator* $_{opt}$ *prefix-expression*
binary-expression → *type-casting-operator*
binary-expressions → *binary-expression* *binary-expressions* $_{opt}$

Assignment Operator

The *assignment operator* sets a new value for a given expression. It has the following form:

`expression` = `value`

The value of the *expression* is set to the value obtained by evaluating the *value*. If the *expression* is a tuple, the *value* must be a tuple with the same

number of elements. (Nested tuples are allowed.) Assignment is performed from each part of the *value* to the corresponding part of the *expression*. For example:

```
(a, _, (b, c)) = ("test", 9.45, (12, 3))
// a is "test", b is 12, c is 3, and 9.45 is ignored
```

The assignment operator doesn't return any value.

GRAMMAR OF AN ASSIGNMENT OPERATOR

assignment-operator → `=`

Ternary Conditional Operator

The *ternary conditional operator* evaluates to one of two given values based on the value of a condition. It has the following form:

`condition` ? `expression used if true` : `expression used if false`

If the *condition* evaluates to `true`, the conditional operator evaluates the first expression and returns its value. Otherwise, it evaluates the second expression and returns its value. The unused expression isn't evaluated.

For an example that uses the ternary conditional operator, see Ternary Conditional Operator.

GRAMMAR OF A CONDITIONAL OPERATOR

conditional-operator → `?` *expression* `:`

Type-Casting Operators

There are four type-casting operators: the `is` operator, the `as` operator, the `as?` operator, and the `as!` operator.

They have the following form:

```
expression is type
expression as type
expression as? type
expression as! type
```

The `is` operator checks at runtime whether the *expression* can be cast to the specified *type*. It returns `true` if the *expression* can be cast to the specified *type*; otherwise, it returns `false`.

The `as` operator performs a cast when it's known at compile time that the cast always succeeds, such as upcasting or bridging. Upcasting lets you use an expression as an instance of its type's supertype, without using an intermediate variable. The following approaches are equivalent:

```swift
func f(_ any: Any) { print("Function for Any") }
func f(_ int: Int) { print("Function for Int") }
let x = 10
f(x)
// Prints "Function for Int"

let y: Any = x
f(y)
// Prints "Function for Any"

f(x as Any)
// Prints "Function for Any"
```

Bridging lets you use an expression of a Swift standard library type such as `String` as its corresponding Foundation type such as `NSString` without needing to create a new instance. For more information on bridging, see [Working with Foundation Types](#).

The `as?` operator performs a conditional cast of the *expression* to the specified *type*. The `as?` operator returns an optional of the specified *type*. At runtime, if the cast succeeds, the value of *expression* is wrapped in an optional and returned; otherwise, the value returned is `nil`. If casting to the specified *type* is guaranteed to fail or is guaranteed to succeed, a compile-time error is raised.

The `as!` operator performs a forced cast of the *expression* to the specified *type*. The `as!` operator returns a value of the specified *type*, not an optional type. If the cast fails, a runtime error is raised. The behavior of `x as! T` is the same as the behavior of `(x as? T)!`.

For more information about type casting and to see examples that use the type-casting operators, see Type Casting.

GRAMMAR OF A TYPE-CASTING OPERATOR

type-casting-operator → **is** *type*
type-casting-operator → **as** *type*
type-casting-operator → **as ?** *type*
type-casting-operator → **as !** *type*

Primary Expressions

Primary expressions are the most basic kind of expression. They can be used as expressions on their own, and they can be combined with other tokens to make prefix expressions, binary expressions, and postfix expressions.

GRAMMAR OF A PRIMARY EXPRESSION

primary-expression → *identifier* *generic-argument-clause* *opt*
primary-expression → *literal-expression*
primary-expression → *self-expression*
primary-expression → *superclass-expression*
primary-expression → *closure-expression*
primary-expression → *parenthesized-expression*
primary-expression → *tuple-expression*
primary-expression → *implicit-member-expression*
primary-expression → *wildcard-expression*
primary-expression → *key-path-expression*
primary-expression → *selector-expression*
primary-expression → *key-path-string-expression*

Literal Expression

A *literal expression* consists of either an ordinary literal (such as a string or a number), an array or dictionary literal, a playground literal, or one of the following special literals:

Literal	Type	Value
`#file`	`String`	The path to the file in which it appears.
`#fileID`	`String`	The name of the file and module in which it appears.
`#filePath`	`String`	The path to the file in which it appears.
`#line`	`Int`	The line number on which it appears.
`#column`	`Int`	The column number in which it begins.
`#function`	`String`	The name of the declaration in which it appears.
`#dsohandle`	`UnsafeRawPointer`	The dynamic shared object (DSO) handle in use where it appears.

The string value of `#file` depends on the language version, to enable migration from the old `#filePath` behavior to the new `#fileID` behavior. Currently, `#file` has the same value as `#filePath`. In a future version of Swift, `#file` will have the same value as `#fileID` instead. To adopt the future behavior, replace `#file` with `#fileID` or `#filePath` as appropriate.

The string value of a `#fileID` expression has the form *module*/*file*, where *file* is the name of the file in which the expression appears and *module* is the name of the module that this file is part of. The string value of a `#filePath`

expression is the full file-system path to the file in which the expression appears. Both of these values can be changed by `#sourceLocation`, as described in [Line Control Statement]. Because `#fileID` doesn't embed the full path to the source file, unlike `#filePath`, it gives you better privacy and reduces the size of the compiled binary. Avoid using `#filePath` outside of tests, build scripts, or other code that doesn't become part of the shipping program.

> NOTE
>
> To parse a `#fileID` expression, read the module name as the text before the first slash (`/`) and the filename as the text after the last slash. In the future, the string might contain multiple slashes, such as `MyModule/some/disambiguation/MyFile.swift`.

Inside a function, the value of `#function` is the name of that function, inside a method it's the name of that method, inside a property getter or setter it's the name of that property, inside special members like `init` or `subscript` it's the name of that keyword, and at the top level of a file it's the name of the current module.

When used as the default value of a function or method parameter, the special literal's value is determined when the default value expression is evaluated at the call site.

```
func logFunctionName(string: String = #function) {
    print(string)
}
func myFunction() {
    logFunctionName() // Prints "myFunction()".
}
```

An *array literal* is an ordered collection of values. It has the following form:

```
[value 1, value 2, ...]
```

The last expression in the array can be followed by an optional comma. The value of an array literal has type `[T]`, where `T` is the type of the expressions inside it. If there are expressions of multiple types, `T` is their closest common supertype. Empty array literals are written using an empty pair of square brackets and can be used to create an empty array of a specified type.

```
var emptyArray: [Double] = []
```

A *dictionary literal* is an unordered collection of key-value pairs. It has the following form:

```
[key 1: value 1, key 2: value 2, ...]
```

The last expression in the dictionary can be followed by an optional comma. The value of a dictionary literal has type `[Key: Value]`, where `Key` is the type of its key expressions and `Value` is the type of its value expressions. If there are expressions of multiple types, `Key` and `Value` are the closest common supertype for their respective values. An empty dictionary literal is written as a colon inside a pair of brackets (`[:]`) to distinguish it from an empty array literal. You can use an empty dictionary literal to create an empty dictionary literal of specified key and value types.

```
var emptyDictionary: [String: Double] = [:]
```

A *playground literal* is used by Xcode to create an interactive representation of a color, file, or image within the program editor. Playground literals in plain text outside of Xcode are represented using a special literal syntax.

For information on using playground literals in Xcode, see [Add a color, file, or image literal] in Xcode Help.

GRAMMAR OF A LITERAL EXPRESSION

literal-expression → *literal*
literal-expression → *array-literal* | *dictionary-literal* | *playground-literal*
literal-expression → **#file** | **#fileID** | **#filePath**
literal-expression → **#line** | **#column** | **#function** | **#dsohandle**
array-literal → **[** *array-literal-items* $_{opt}$ **]**
array-literal-items → *array-literal-item* **,** $_{opt}$ | *array-literal-item* **,** *array-literal-items*
array-literal-item → *expression*
dictionary-literal → **[** *dictionary-literal-items* **]** | **[:]**
dictionary-literal-items → *dictionary-literal-item* **,** $_{opt}$ | *dictionary-literal-item* **,** *dictionary-literal-items*
dictionary-literal-item → *expression* **:** *expression*
playground-literal → **#colorLiteral (red :** *expression* **, green :** *expression* **, blue :** *expression* **, alpha :** *expression* **)**
playground-literal → **#fileLiteral (resourceName :** *expression* **)**
playground-literal → **#imageLiteral (resourceName :** *expression* **)**

Self Expression

The `self` expression is an explicit reference to the current type or instance of the type in which it occurs. It has the following forms:

```
self
self.member name
self[subscript index]
self(initializer arguments)
self.init(initializer arguments)
```

In an initializer, subscript, or instance method, `self` refers to the current instance of the type in which it occurs. In a type method, `self` refers to the current type in which it occurs.

The `self` expression is used to specify scope when accessing members, providing disambiguation when there's another variable of the same name in scope, such as a function parameter. For example:

```
class SomeClass {
    var greeting: String
    init(greeting: String) {
        self.greeting = greeting
    }
}
```

In a mutating method of a value type, you can assign a new instance of that value type to `self`. For example:

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY:
     Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

GRAMMAR OF A SELF EXPRESSION

self-expression → **self** | *self-method-expression* | *self-subscript-expression* | *self-initializer-expression*
self-method-expression → **self** **.** *identifier*
self-subscript-expression → **self** **[** *function-call-argument-list* **]**
self-initializer-expression → **self** **.** **init**

Superclass Expression

A *superclass expression* lets a class interact with its superclass. It has one of the following forms:

```
super.member name
super[subscript index]
super.init(initializer arguments)
```

The first form is used to access a member of the superclass. The second form is used to access the superclass's subscript implementation. The third form is used to access an initializer of the superclass.

Subclasses can use a superclass expression in their implementation of members, subscripting, and initializers to make use of the implementation in their superclass.

GRAMMAR OF A SUPERCLASS EXPRESSION

superclass-expression → *superclass-method-expression* | *superclass-subscript-expression* | *superclass-initializer-expression*
superclass-method-expression → **super** . *identifier*
superclass-subscript-expression → **super** [*function-call-argument-list*]
superclass-initializer-expression → **super** . **init**

Closure Expression

A *closure expression* creates a closure, also known as a *lambda* or an *anonymous function* in other programming languages. Like a function declaration, a closure contains statements, and it captures constants and variables from its enclosing scope. It has the following form:

```
{ (parameters) -> return type in
    statements
}
```

The *parameters* have the same form as the parameters in a function declaration, as described in Function Declaration.

There are several special forms that allow closures to be written more concisely:

- A closure can omit the types of its parameters, its return type, or both. If you omit the parameter names and both types, omit the `in` keyword before the statements. If the omitted types can't be inferred, a compile-time error is raised.

- A closure may omit names for its parameters. Its parameters are then implicitly named `$` followed by their position: `$0`, `$1`, `$2`, and so on.

- A closure that consists of only a single expression is understood to return the value of that expression. The contents of this expression are also considered when performing type inference on the surrounding expression.

The following closure expressions are equivalent:

```
myFunction { (x: Int, y: Int) -> Int in
    return x + y
}

myFunction { x, y in
    return x + y
}

myFunction { return $0 + $1 }

myFunction { $0 + $1 }
```

For information about passing a closure as an argument to a function, see [Function Call Expression](#).

Closure expressions can be used without being stored in a variable or constant, such as when you immediately use a closure as part of a function call. The closure expressions passed to `myFunction` in code above are examples of this kind of immediate use. As a result, whether a closure expression is escaping or nonescaping depends on the surrounding context of the expression. A closure expression is nonescaping if it's called immediately or passed as a nonescaping function argument. Otherwise, the closure expression is escaping.

For more information about escaping closures, see Escaping Closures.

Capture Lists

By default, a closure expression captures constants and variables from its surrounding scope with strong references to those values. You can use a *capture list* to explicitly control how values are captured in a closure.

A capture list is written as a comma-separated list of expressions surrounded by square brackets, before the list of parameters. If you use a capture list, you must also use the `in` keyword, even if you omit the parameter names, parameter types, and return type.

The entries in the capture list are initialized when the closure is created. For each entry in the capture list, a constant is initialized to the value of the constant or variable that has the same name in the surrounding scope. For example in the code below, `a` is included in the capture list but `b` is not, which gives them different behavior.

```
var a = 0
var b = 0
let closure = { [a] in
    print(a, b)
}

a = 10
b = 10
closure()
// Prints "0 10"
```

There are two different things named `a`, the variable in the surrounding scope and the constant in the closure's scope, but only one variable named `b`. The `a` in the inner scope is initialized with the value of the `a` in the outer scope when the closure is created, but their values aren't connected in any special way. This means that a change to the value of `a` in the outer scope doesn't affect the value of `a` in the inner scope, nor does a change to `a` inside the closure affect the value of `a` outside the closure. In contrast, there's only one variable named `b`—the `b` in the outer scope—so changes from inside or outside the closure are visible in both places.

This distinction isn't visible when the captured variable's type has reference semantics. For example, there are two things named `x` in the code below, a variable in the outer scope and a constant in the inner scope, but they both refer to the same object because of reference semantics.

```
class SimpleClass {
    var value: Int = 0
}
var x = SimpleClass()
var y = SimpleClass()
let closure = { [x] in
    print(x.value, y.value)
}

x.value = 10
y.value = 10
closure()
// Prints "10 10"
```

If the type of the expression's value is a class, you can mark the expression in a capture list with `weak` or `unowned` to capture a weak or unowned reference to the expression's value.

```
myFunction { print(self.title) }
     // implicit strong capture
myFunction { [self] in print(self.title) }
     // explicit strong capture
myFunction { [weak self] in print(self!.title) }
     // weak capture
myFunction { [unowned self] in print(self.title) }
     // unowned capture
```

You can also bind an arbitrary expression to a named value in a capture list. The expression is evaluated when the closure is created, and the value is captured with the specified strength. For example:

```
// Weak capture of "self.parent" as "parent"
myFunction { [weak parent = self.parent] in
     print(parent!.title) }
```

For more information and examples of closure expressions, see Closure Expressions. For more information and examples of capture lists, see Resolving Strong Reference Cycles for Closures.

GRAMMAR OF A CLOSURE EXPRESSION

closure-expression → **{** *closure-signature* *opt* *statements* *opt* **}**
closure-signature → *capture-list* *opt* *closure-parameter-clause* **throws** *opt* *function-result* *opt* **in**
closure-signature → *capture-list* **in**
closure-parameter-clause → **( )** | **(** *closure-parameter-list* **)** | *identifier-list*
closure-parameter-list → *closure-parameter* | *closure-parameter* **,** *closure-parameter-list*
closure-parameter → *closure-parameter-name* *type-annotation* *opt*
closure-parameter → *closure-parameter-name* *type-annotation* **...**
closure-parameter-name → *identifier*
capture-list → **[** *capture-list-items* **]**
capture-list-items → *capture-list-item* | *capture-list-item* **,** *capture-list-items*
capture-list-item → *capture-specifier* *opt* *identifier*
capture-list-item → *capture-specifier* *opt* *identifier* **=** *expression*
capture-list-item → *capture-specifier* *opt* *self-expression*
capture-specifier → **weak** | **unowned** | **unowned(safe)** | **unowned(unsafe)**

Implicit Member Expression

An *implicit member expression* is an abbreviated way to access a member of a type, such as an enumeration case or a type method, in a context where

type inference can determine the implied type. It has the following form:

```
.member name
```

For example:

```
var x = MyEnumeration.someValue
x = .anotherValue
```

If the inferred type is an optional, you can also use a member of the non-optional type in an implicit member expression.

```
var someOptional: MyEnumeration? = .someValue
```

Implicit member expressions can be followed by a postfix operator or other postfix syntax listed in Postfix Expressions. This is called a *chained implicit member expression*. Although it's common for all of the chained postfix expressions to have the same type, the only requirement is that the whole chained implicit member expression needs to be convertible to the type implied by its context. Specifically, if the implied type is an optional you can use a value of the non-optional type, and if the implied type is a class type you can use a value of one of its subclasses. For example:

```
class SomeClass {
    static var shared = SomeClass()
    static var sharedSubclass = SomeSubclass()
    var a = AnotherClass()
}
class SomeSubclass: SomeClass { }
class AnotherClass {
    static var s = SomeClass()
    func f() -> SomeClass { return AnotherClass.s }
}
let x: SomeClass = .shared.a.f()
let y: SomeClass? = .shared
let z: SomeClass = .sharedSubclass
```

In the code above, the type of `x` matches the type implied by its context exactly, the type of `y` is convertible from `SomeClass` to `SomeClass?`, and the type of `z` is convertible from `SomeSubclass` to `SomeClass`.

GRAMMAR OF A IMPLICIT MEMBER EXPRESSION

implicit-member-expression → . *identifier*
implicit-member-expression → . *identifier* . *postfix-expression*

Parenthesized Expression

A *parenthesized expression* consists of an expression surrounded by parentheses. You can use parentheses to specify the precedence of operations by explicitly grouping expressions. Grouping parentheses don't change an expression's type—for example, the type of `(1)` is simply `Int`.

GRAMMAR OF A PARENTHESIZED EXPRESSION

parenthesized-expression → (*expression*)

Tuple Expression

A *tuple expression* consists of a comma-separated list of expressions surrounded by parentheses. Each expression can have an optional identifier before it, separated by a colon (`:`). It has the following form:

```
(identifier 1: expression 1, identifier 2:
      expression 2, ...)
```

Each identifier in a tuple expression must be unique within the scope of the tuple expression. In a nested tuple expression, identifiers at the same level of nesting must be unique. For example, `(a: 10, a: 20)` is invalid because the label `a` appears twice at the same level. However, `(a: 10, b: (a: 1, x: 2))` is valid—although `a` appears twice, it appears once in the outer tuple and once in the inner tuple.

A tuple expression can contain zero expressions, or it can contain two or more expressions. A single expression inside parentheses is a parenthesized expression.

> NOTE
>
> Both an empty tuple expression and an empty tuple type are written `()` in Swift. Because `Void` is a type alias for `()`, you can use it to write an empty tuple type. However, like all type aliases, `Void` is always a type—you can't use it to write an empty tuple expression.

GRAMMAR OF A TUPLE EXPRESSION

tuple-expression → ( ) | (*tuple-element* , *tuple-element-list*)
tuple-element-list → *tuple-element* | *tuple-element* , *tuple-element-list*
tuple-element → *expression* | *identifier* : *expression*

Wildcard Expression

A *wildcard expression* is used to explicitly ignore a value during an assignment. For example, in the following assignment 10 is assigned to `x` and 20 is ignored:

```
(x, _) = (10, 20)
// x is 10, and 20 is ignored
```

GRAMMAR OF A WILDCARD EXPRESSION

wildcard-expression → `_`

Key-Path Expression

A *key-path expression* refers to a property or subscript of a type. You use key-path expressions in dynamic programming tasks, such as key-value observing. They have the following form:

```
\type name.path
```

The *type name* is the name of a concrete type, including any generic parameters, such as `String`, `[Int]`, or `Set<Int>`.

The *path* consists of property names, subscripts, optional-chaining expressions, and forced unwrapping expressions. Each of these key-path components can be repeated as many times as needed, in any order.

At compile time, a key-path expression is replaced by an instance of the `KeyPath` class.

To access a value using a key path, pass the key path to the `subscript(keyPath:)` subscript, which is available on all types. For example:

```
struct SomeStructure {
    var someValue: Int
}

let s = SomeStructure(someValue: 12)
let pathToProperty = \SomeStructure.someValue

let value = s[keyPath: pathToProperty]
// value is 12
```

The *type name* can be omitted in contexts where type inference can determine the implied type. The following code uses `\.someProperty` instead of `\SomeClass.someProperty`:

```
class SomeClass: NSObject {
    @objc dynamic var someProperty: Int
    init(someProperty: Int) {
        self.someProperty = someProperty
    }
}

let c = SomeClass(someProperty: 10)
c.observe(\.someProperty) { object, change in
    // ...
}
```

The *path* can refer to `self` to create the identity key path (`\.self`). The identity key path refers to a whole instance, so you can use it to access and

change all of the data stored in a variable in a single step. For example:

```
var compoundValue = (a: 1, b: 2)
// Equivalent to compoundValue = (a: 10, b: 20)
compoundValue[keyPath: \.self] = (a: 10, b: 20)
```

The *path* can contain multiple property names, separated by periods, to refer to a property of a property's value. This code uses the key path expression `\OuterStructure.outer.someValue` to access the `someValue` property of the `OuterStructure` type's `outer` property:

```
struct OuterStructure {
    var outer: SomeStructure
    init(someValue: Int) {
        self.outer = SomeStructure(someValue:
     someValue)
    }
}

let nested = OuterStructure(someValue: 24)
let nestedKeyPath = \OuterStructure.outer.someValue

let nestedValue = nested[keyPath: nestedKeyPath]
// nestedValue is 24
```

The *path* can include subscripts using brackets, as long as the subscript's parameter type conforms to the `Hashable` protocol. This example uses a subscript in a key path to access the second element of an array:

```
1  let greetings = ["hello", "hola", "bonjour", "안녕"]
2  let myGreeting = greetings[keyPath: \[String].[1]]
3  // myGreeting is 'hola'
```

The value used in a subscript can be a named value or a literal. Values are captured in key paths using value semantics. The following code uses the variable `index` in both a key-path expression and in a closure to access the third element of the `greetings` array. When `index` is modified, the key-path expression still references the third element, while the closure uses the new index.

```
var index = 2
let path = \[String].[index]
let fn: ([String]) -> String = { strings in
     strings[index] }

print(greetings[keyPath: path])
// Prints "bonjour"
print(fn(greetings))
// Prints "bonjour"

// Setting 'index' to a new value doesn't affect
     'path'
index += 1
print(greetings[keyPath: path])
// Prints "bonjour"

// Because 'fn' closes over 'index', it uses the new
     value
print(fn(greetings))
// Prints "안녕"
```

The *path* can use optional chaining and forced unwrapping. This code uses optional chaining in a key path to access a property of an optional string:

```
let firstGreeting: String? = greetings.first
print(firstGreeting?.count as Any)
// Prints "Optional(5)"

// Do the same thing using a key path.
let count = greetings[keyPath: \
     [String].first?.count]
print(count as Any)
// Prints "Optional(5)"
```

You can mix and match components of key paths to access values that are deeply nested within a type. The following code accesses different values and properties of a dictionary of arrays by using key-path expressions that combine these components.

```
let interestingNumbers = ["prime": [2, 3, 5, 7, 11, 13, 17],
                          "triangular": [1, 3, 6, 10, 15, 21, 28],
                          "hexagonal": [1, 6, 15, 28, 45, 66, 91]]
print(interestingNumbers[keyPath: \[String: [Int]].["prime"]] as Any)
// Prints "Optional([2, 3, 5, 7, 11, 13, 17])"
print(interestingNumbers[keyPath: \[String: [Int]].["prime"]![0]])
// Prints "2"
print(interestingNumbers[keyPath: \[String: [Int]].["hexagonal"]!.count])
// Prints "7"
print(interestingNumbers[keyPath: \[String: [Int]].["hexagonal"]!.count.bitWidth])
// Prints "64"
```

You can use a key path expression in contexts where you would normally provide a function or closure. Specifically, you can use a key path expression whose root type is `SomeType` and whose path produces a value of type `Value`, instead of a function or closure of type `(SomeType) -> Value`.

```swift
struct Task {
    var description: String
    var completed: Bool
}
var toDoList = [
    Task(description: "Practice ping-pong.", completed: false),
    Task(description: "Buy a pirate costume.", completed: true),
    Task(description: "Visit Boston in the Fall.", completed: false),
]

// Both approaches below are equivalent.
let descriptions = toDoList.filter(\.completed).map(\.description)
let descriptions2 = toDoList.filter { $0.completed }.map { $0.description }
```

Any side effects of a key path expression are evaluated only at the point where the expression is evaluated. For example, if you make a function call inside a subscript in a key path expression, the function is called only once as part of evaluating the expression, not every time the key path is used.

```swift
func makeIndex() -> Int {
    print("Made an index")
    return 0
}
// The line below calls makeIndex().
let taskKeyPath = \[Task][makeIndex()]
// Prints "Made an index"

// Using taskKeyPath doesn't call makeIndex() again.
let someTask = toDoList[keyPath: taskKeyPath]
```

For more information about using key paths in code that interacts with Objective-C APIs, see Using Objective-C Runtime Features in Swift. For information about key-value coding and key-value observing, see Key-Value Coding Programming Guide and Key-Value Observing Programming Guide.

GRAMMAR OF A KEY-PATH EXPRESSION

key-path-expression → \ *type* $_{opt}$. *key-path-components*
key-path-components → *key-path-component* | *key-path-component* . *key-path-components*
key-path-component → *identifier* *key-path-postfixes* $_{opt}$ | *key-path-postfixes*
key-path-postfixes → *key-path-postfix* *key-path-postfixes* $_{opt}$
key-path-postfix → **?** | **!** | **self** | **[** *function-call-argument-list* **]**

Selector Expression

A selector expression lets you access the selector used to refer to a method or to a property's getter or setter in Objective-C. It has the following form:

```
#selector(method name)
#selector(getter: property name)
#selector(setter: property name)
```

The *method name* and *property name* must be a reference to a method or a property that's available in the Objective-C runtime. The value of a selector expression is an instance of the `Selector` type. For example:

```
class SomeClass: NSObject {
    @objc let property: String

    @objc(doSomethingWithInt:)
    func doSomething(_ x: Int) { }

    init(property: String) {
        self.property = property
    }
}
let selectorForMethod =
     #selector(SomeClass.doSomething(_:))
let selectorForPropertyGetter = #selector(getter:
     SomeClass.property)
```

When creating a selector for a property's getter, the *property name* can be a reference to a variable or constant property. In contrast, when creating a selector for a property's setter, the *property name* must be a reference to a variable property only.

The *method name* can contain parentheses for grouping, as well the `as` operator to disambiguate between methods that share a name but have different type signatures. For example:

```
extension SomeClass {
    @objc(doSomethingWithString:)
    func doSomething(_ x: String) { }
}
let anotherSelector =
     #selector(SomeClass.doSomething(_:) as
     (SomeClass) -> (String) -> Void)
```

Because a selector is created at compile time, not at runtime, the compiler can check that a method or property exists and that they're exposed to the Objective-C runtime.

> NOTE
>
> Although the *method name* and the *property name* are expressions, they're never evaluated.

For more information about using selectors in Swift code that interacts with Objective-C APIs, see [Using Objective-C Runtime Features in Swift](#).

GRAMMAR OF A SELECTOR EXPRESSION

selector-expression → **#selector** (*expression*)
selector-expression → **#selector** (**getter:** *expression*)
selector-expression → **#selector** (**setter:** *expression*)

Key-Path String Expression

A key-path string expression lets you access the string used to refer to a property in Objective-C, for use in key-value coding and key-value

observing APIs. It has the following form:

```
#keyPath(property name)
```

The *property name* must be a reference to a property that's available in the Objective-C runtime. At compile time, the key-path string expression is replaced by a string literal. For example:

```
class SomeClass: NSObject {
    @objc var someProperty: Int
    init(someProperty: Int) {
        self.someProperty = someProperty
    }
}

let c = SomeClass(someProperty: 12)
let keyPath = #keyPath(SomeClass.someProperty)

if let value = c.value(forKey: keyPath) {
    print(value)
}
// Prints "12"
```

When you use a key-path string expression within a class, you can refer to a property of that class by writing just the property name, without the class name.

```
extension SomeClass {
    func getSomeKeyPath() -> String {
        return #keyPath(someProperty)
    }
}
print(keyPath == c.getSomeKeyPath())
// Prints "true"
```

Because the key path string is created at compile time, not at runtime, the compiler can check that the property exists and that the property is exposed to the Objective-C runtime.

For more information about using key paths in Swift code that interacts with Objective-C APIs, see Using Objective-C Runtime Features in Swift. For information about key-value coding and key-value observing, see Key-Value Coding Programming Guide and Key-Value Observing Programming Guide.

> NOTE
> Although the *property name* is an expression, it's never evaluated.

GRAMMAR OF A KEY-PATH STRING EXPRESSION

key-path-string-expression → **#keyPath** (*expression*)

Postfix Expressions

Postfix expressions are formed by applying a postfix operator or other postfix syntax to an expression. Syntactically, every primary expression is also a postfix expression.

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

For information about the operators provided by the Swift standard library, see Operator Declarations.

GRAMMAR OF A POSTFIX EXPRESSION

postfix-expression → *primary-expression*
postfix-expression → *postfix-expression* *postfix-operator*
postfix-expression → *function-call-expression*
postfix-expression → *initializer-expression*
postfix-expression → *explicit-member-expression*
postfix-expression → *postfix-self-expression*
postfix-expression → *subscript-expression*
postfix-expression → *forced-value-expression*
postfix-expression → *optional-chaining-expression*

Function Call Expression

A *function call expression* consists of a function name followed by a comma-separated list of the function's arguments in parentheses. Function call expressions have the following form:

```
function name(argument value 1,
              argument value 2)
```

The *function name* can be any expression whose value is of a function type.

If the function definition includes names for its parameters, the function call must include names before its argument values, separated by a colon (`:`). This kind of function call expression has the following form:

```
function name(argument name 1: argument value 1,
              argument name 2: argument value 2)
```

A function call expression can include trailing closures in the form of closure expressions immediately after the closing parenthesis. The trailing closures are understood as arguments to the function, added after the last parenthesized argument. The first closure expression is unlabeled; any additional closure expressions are preceded by their argument labels. The example below shows the equivalent version of function calls that do and don't use trailing closure syntax:

```
// someFunction takes an integer and a closure as its arguments
someFunction(x: x, f: { $0 == 13 })
someFunction(x: x) { $0 == 13 }

// anotherFunction takes an integer and two closures as its arguments
anotherFunction(x: x, f: { $0 == 13 }, g: { print(99) })
anotherFunction(x: x) { $0 == 13 } g: { print(99) }
```

If the trailing closure is the function's only argument, you can omit the parentheses.

```
// someMethod takes a closure as its only argument
myData.someMethod() { $0 == 13 }
myData.someMethod { $0 == 13 }
```

To include the trailing closures in the arguments, the compiler examines the function's parameters from left to right as follows:

Trailing Closure	Parameter	Action
Labeled	Labeled	If the labels are the same, the closure matches the parameter; otherwise, the parameter is skipped.
Labeled	Unlabeled	The parameter is skipped.
Unlabeled	Labeled or unlabeled	If the parameter structurally resembles a function type, as defined below, the closure matches the parameter; otherwise, the parameter is skipped.

The trailing closure is passed as the argument for the parameter that it matches. Parameters that were skipped during the scanning process don't have an argument passed to them—for example, they can use a default parameter. After finding a match, scanning continues with the next trailing closure and the next parameter. At the end of the matching process, all trailing closures must have a match.

A parameter *structurally resembles* a function type if the parameter isn't an in-out parameter, and the parameter is one of the following:

- A parameter whose type is a function type, like `(Bool) -> Int`
- An autoclosure parameter whose wrapped expression's type is a function type, like `@autoclosure () -> ((Bool) -> Int)`
- A variadic parameter whose array element type is a function type, like `((Bool) -> Int)...`

- A parameter whose type is wrapped in one or more layers of optional, like `Optional<(Bool) -> Int>`
- A parameter whose type combines these allowed types, like `(Optional<(Bool) -> Int>)...`

When a trailing closure is matched to a parameter whose type structurally resembles a function type, but isn't a function, the closure is wrapped as needed. For example, if the parameter's type is an optional type, the closure is wrapped in `Optional` automatically.

To ease migration of code from versions of Swift prior to 5.3—which performed this matching from right to left—the compiler checks both the left-to-right and right-to-left orderings. If the scan directions produce different results, the old right-to-left ordering is used and the compiler generates a warning. A future version of Swift will always use the left-to-right ordering.

```
typealias Callback = (Int) -> Int
func someFunction(firstClosure: Callback? = nil,
                  secondClosure: Callback? = nil) {
    let first = firstClosure?(10)
    let second = secondClosure?(20)
    print(first ?? "–", second ?? "–")
}

someFunction()  // Prints "– –"
someFunction { return $0 + 100 }  // Ambiguous
someFunction { return $0 } secondClosure: { return
     $0 }  // Prints "10 20"
```

In the example above, the function call marked “Ambiguous” prints “- 120” and produces a compiler warning on Swift 5.3. A future version of Swift will print “110 -”.

A class, structure, or enumeration type can enable syntactic sugar for function call syntax by declaring one of several methods, as described in Methods with Special Names.

Implicit Conversion to a Pointer Type

In a function call expression, if the argument and parameter have a different type, the compiler tries to make their types match by applying one of the implicit conversions in the following list:

- `inout SomeType` can become `UnsafePointer<SomeType>` or `UnsafeMutablePointer<SomeType>`
- `inout Array<SomeType>` can become `UnsafePointer<SomeType>` or `UnsafeMutablePointer<SomeType>`
- `Array<SomeType>` can become `UnsafePointer<SomeType>`
- `String` can become `UnsafePointer<CChar>`

The following two function calls are equivalent:

```swift
func unsafeFunction(pointer: UnsafePointer<Int>) {
    // ...
}
var myNumber = 1234

unsafeFunction(pointer: &myNumber)
withUnsafePointer(to: myNumber) {
     unsafeFunction(pointer: $0) }
```

A pointer that's created by these implicit conversions is valid only for the duration of the function call. To avoid undefined behavior, ensure that your code never persists the pointer after the function call ends.

> NOTE
>
> When implicitly converting an array to an unsafe pointer, Swift ensures that the array's storage is contiguous by converting or copying the array as needed. For example, you can use this syntax with an array that was bridged to `Array` from an `NSArray` subclass that makes no API contract about its storage. If you need to guarantee that the array's storage is already contiguous, so the implicit conversion never needs to do this work, use `ContiguousArray` instead of `Array`.

Using `&` instead of an explicit function like `withUnsafePointer(to:)` can help make calls to low-level C functions more readable, especially when the function takes several pointer arguments. However, when calling functions from other Swift code, avoid using `&` instead of using the unsafe APIs explicitly.

GRAMMAR OF A FUNCTION CALL EXPRESSION

function-call-expression → *postfix-expression* *function-call-argument-clause*
function-call-expression → *postfix-expression* *function-call-argument-clause* *opt* *trailing-closures*
function-call-argument-clause → **(** **)** | **(** *function-call-argument-list* **)**
function-call-argument-list → *function-call-argument* | *function-call-argument* **,** *function-call-argument-list*
function-call-argument → *expression* | *identifier* **:** *expression*
function-call-argument → *operator* | *identifier* **:** *operator*
trailing-closures → *closure-expression* *labeled-trailing-closures* *opt*
labeled-trailing-closures → *labeled-trailing-closure* *labeled-trailing-closures* *opt*
labeled-trailing-closure → *identifier* **:** *closure-expression*

Initializer Expression

An *initializer expression* provides access to a type's initializer. It has the following form:

```
expression.init(initializer arguments)
```

You use the initializer expression in a function call expression to initialize a new instance of a type. You also use an initializer expression to delegate to the initializer of a superclass.

```
class SomeSubClass: SomeSuperClass {
    override init() {
        // subclass initialization goes here
        super.init()
    }
}
```

Like a function, an initializer can be used as a value. For example:

```
// Type annotation is required because String has multiple initializers.
let initializer: (Int) -> String = String.init
let oneTwoThree = [1, 2, 3].map(initializer).reduce("", +)
print(oneTwoThree)
// Prints "123"
```

If you specify a type by name, you can access the type's initializer without using an initializer expression. In all other cases, you must use an initializer expression.

```
let s1 = SomeType.init(data: 3)  // Valid
let s2 = SomeType(data: 1)       // Also valid

let s3 = type(of: someValue).init(data: 7)  // Valid
let s4 = type(of: someValue)(data: 5)       // Error
```

GRAMMAR OF AN INITIALIZER EXPRESSION

initializer-expression → *postfix-expression* **. init**
initializer-expression → *postfix-expression* **. init (*argument-names*)**

Explicit Member Expression

An *explicit member expression* allows access to the members of a named type, a tuple, or a module. It consists of a period (`.`) between the item and the identifier of its member.

`expression`**.**`member name`

The members of a named type are named as part of the type's declaration or extension. For example:

```
class SomeClass {
    var someProperty = 42
}
let c = SomeClass()
let y = c.someProperty  // Member access
```

The members of a tuple are implicitly named using integers in the order they appear, starting from zero. For example:

```
var t = (10, 20, 30)
t.0 = t.1
// Now t is (20, 20, 30)
```

The members of a module access the top-level declarations of that module.

Types declared with the `dynamicMemberLookup` attribute include members that are looked up at runtime, as described in Attributes.

To distinguish between methods or initializers whose names differ only by the names of their arguments, include the argument names in parentheses, with each argument name followed by a colon (`:`). Write an underscore (_) for an argument with no name. To distinguish between overloaded methods, use a type annotation. For example:

```
class SomeClass {
    func someMethod(x: Int, y: Int) {}
    func someMethod(x: Int, z: Int) {}
    func overloadedMethod(x: Int, y: Int) {}
    func overloadedMethod(x: Int, y: Bool) {}
}
let instance = SomeClass()

let a = instance.someMethod              // Ambiguous
let b = instance.someMethod(x:y:)        // Unambiguous

let d = instance.overloadedMethod        // Ambiguous
let d = instance.overloadedMethod(x:y:)  // Still ambiguous
let d: (Int, Bool) -> Void  = instance.overloadedMethod(x:y:)  // Unambiguous
```

If a period appears at the beginning of a line, it's understood as part of an explicit member expression, not as an implicit member expression. For example, the following listing shows chained method calls split over several lines:

```
let x = [10, 3, 20, 15, 4]
    .sorted()
    .filter { $0 > 5 }
    .map { $0 * 100 }
```

GRAMMAR OF AN EXPLICIT MEMBER EXPRESSION

explicit-member-expression → *postfix-expression* **.** *decimal-digits*
explicit-member-expression → *postfix-expression* **.** *identifier* *generic-argument-clause* *opt*
explicit-member-expression → *postfix-expression* **.** *identifier* **(** *argument-names* **)**
argument-names → *argument-name* *argument-names* *opt*
argument-name → *identifier* **:**

Postfix Self Expression

A postfix `self` expression consists of an expression or the name of a type, immediately followed by `.self`. It has the following forms:

```
expression.self
type.self
```

The first form evaluates to the value of the *expression*. For example, `x.self` evaluates to `x`.

The second form evaluates to the value of the *type*. Use this form to access a type as a value. For example, because `SomeClass.self` evaluates to the `SomeClass` type itself, you can pass it to a function or method that accepts a type-level argument.

GRAMMAR OF A POSTFIX SELF EXPRESSION

postfix-self-expression → *postfix-expression* **. self**

Subscript Expression

A *subscript expression* provides subscript access using the getter and setter of the corresponding subscript declaration. It has the following form:

```
expression[index expressions]
```

To evaluate the value of a subscript expression, the subscript getter for the *expression*'s type is called with the *index expressions* passed as the subscript parameters. To set its value, the subscript setter is called in the same way.

For information about subscript declarations, see Protocol Subscript Declaration.

GRAMMAR OF A SUBSCRIPT EXPRESSION

subscript-expression → *postfix-expression* **[** *function-call-argument-list* **]**

Forced-Value Expression

A *forced-value expression* unwraps an optional value that you are certain isn't `nil`. It has the following form:

```
expression!
```

If the value of the *expression* isn't `nil`, the optional value is unwrapped and returned with the corresponding non-optional type. Otherwise, a runtime error is raised.

The unwrapped value of a forced-value expression can be modified, either by mutating the value itself, or by assigning to one of the value's members. For example:

```
var x: Int? = 0
x! += 1
// x is now 1

var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]
someDictionary["a"]![0] = 100
// someDictionary is now ["a": [100, 2, 3], "b":
     [10, 20]]
```

GRAMMAR OF A FORCED-VALUE EXPRESSION

forced-value-expression → *postfix-expression* !

Optional-Chaining Expression

An *optional-chaining expression* provides a simplified syntax for using optional values in postfix expressions. It has the following form:

`expression`?

The postfix `?` operator makes an optional-chaining expression from an expression without changing the expression's value.

Optional-chaining expressions must appear within a postfix expression, and they cause the postfix expression to be evaluated in a special way. If the value of the optional-chaining expression is `nil`, all of the other operations in the postfix expression are ignored and the entire postfix expression evaluates to `nil`. If the value of the optional-chaining expression isn't `nil`, the value of the optional-chaining expression is unwrapped and used to evaluate the rest of the postfix expression. In either case, the value of the postfix expression is still of an optional type.

If a postfix expression that contains an optional-chaining expression is nested inside other postfix expressions, only the outermost expression returns an optional type. In the example below, when `c` isn't `nil`, its value is unwrapped and used to evaluate `.property`, the value of which is used to evaluate `.performAction()`. The entire expression `c?.property.performAction()` has a value of an optional type.

```
var c: SomeClass?
var result: Bool? = c?.property.performAction()
```

The following example shows the behavior of the example above without using optional chaining.

```
var result: Bool?
if let unwrappedC = c {
    result = unwrappedC.property.performAction()
}
```

The unwrapped value of an optional-chaining expression can be modified, either by mutating the value itself, or by assigning to one of the value's members. If the value of the optional-chaining expression is `nil`, the expression on the right-hand side of the assignment operator isn't evaluated. For example:

```
func someFunctionWithSideEffects() -> Int {
    return 42  // No actual side effects.
}
var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]

someDictionary["not here"]?[0] = someFunctionWithSideEffects()
// someFunctionWithSideEffects isn't evaluated
// someDictionary is still ["a": [1, 2, 3], "b": [10, 20]]

someDictionary["a"]?[0] = someFunctionWithSideEffects()
// someFunctionWithSideEffects is evaluated and returns 42
// someDictionary is now ["a": [42, 2, 3], "b": [10, 20]]
```

GRAMMAR OF AN OPTIONAL-CHAINING EXPRESSION

optional-chaining-expression → *postfix-expression* **?**

Statements

In Swift, there are three kinds of statements: simple statements, compiler control statements, and control flow statements. Simple statements are the most common and consist of either an expression or a declaration. Compiler control statements allow the program to change aspects of the compiler's behavior and include a conditional compilation block and a line control statement.

Control flow statements are used to control the flow of execution in a program. There are several types of control flow statements in Swift, including loop statements, branch statements, and control transfer statements. Loop statements allow a block of code to be executed repeatedly, branch statements allow a certain block of code to be executed only when certain conditions are met, and control transfer statements provide a way to alter the order in which code is executed. In addition, Swift provides a `do` statement to introduce scope, and catch and handle errors, and a `defer` statement for running cleanup actions just before the current scope exits.

A semicolon (`;`) can optionally appear after any statement and is used to separate multiple statements if they appear on the same line.

GRAMMAR OF A STATEMENT

statement → *expression* ;*opt*
statement → *declaration* ;*opt*
statement → *loop-statement* ;*opt*
statement → *branch-statement* ;*opt*
statement → *labeled-statement* ;*opt*
statement → *control-transfer-statement* ;*opt*
statement → *defer-statement* ;*opt*
statement → *do-statement* ;*opt*
statement → *compiler-control-statement*
statements → *statement* *statements* *opt*

Loop Statements

Loop statements allow a block of code to be executed repeatedly, depending on the conditions specified in the loop. Swift has three loop statements: a `for-in` statement, a `while` statement, and a `repeat-while` statement.

Control flow in a loop statement can be changed by a `break` statement and a `continue` statement and is discussed in Break Statement and Continue Statement below.

GRAMMAR OF A LOOP STATEMENT

loop-statement → *for-in-statement*
loop-statement → *while-statement*
loop-statement → *repeat-while-statement*

For-In Statement

A `for-in` statement allows a block of code to be executed once for each item in a collection (or any type) that conforms to the `Sequence` protocol.

A `for-in` statement has the following form:

```
for item in collection {
    statements
}
```

The `makeIterator()` method is called on the *collection* expression to obtain a value of an iterator type—that is, a type that conforms to the `IteratorProtocol` protocol. The program begins executing a loop by calling the `next()` method on the iterator. If the value returned isn't `nil`, it's assigned to the *item* pattern, the program executes the *statements*, and then continues execution at the beginning of the loop. Otherwise, the program doesn't perform assignment or execute the *statements*, and it's finished executing the `for-in` statement.

GRAMMAR OF A FOR-IN STATEMENT

for-in-statement → **for case**$_{opt}$ *pattern* **in** *expression* *where-clause*$_{opt}$ *code-block*

While Statement

A `while` statement allows a block of code to be executed repeatedly, as long as a condition remains true.

A `while` statement has the following form:

```
while condition {
    statements
}
```

A `while` statement is executed as follows:

1. The *condition* is evaluated.

   If `true`, execution continues to step 2. If `false`, the program is finished executing the `while` statement.

2. The program executes the *statements*, and execution returns to step 1.

Because the value of the *condition* is evaluated before the *statements* are executed, the *statements* in a `while` statement can be executed zero or more times.

The value of the *condition* must be of type `Bool` or a type bridged to `Bool`. The condition can also be an optional binding declaration, as discussed in Optional Binding.

GRAMMAR OF A WHILE STATEMENT

while-statement → **while** *condition-list* *code-block*
condition-list → *condition* | *condition* , *condition-list*
condition → *expression* | *availability-condition* | *case-condition* | *optional-binding-condition*
case-condition → **case** *pattern* *initializer*
optional-binding-condition → **let** *pattern* *initializer* | **var** *pattern* *initializer*

Repeat-While Statement

A `repeat-while` statement allows a block of code to be executed one or more times, as long as a condition remains true.

A `repeat-while` statement has the following form:

```
repeat {
    statements
} while condition
```

A `repeat-while` statement is executed as follows:

1. The program executes the *statements*, and execution continues to step 2.

2. The *condition* is evaluated.

   If `true`, execution returns to step 1. If `false`, the program is finished executing the `repeat-while` statement.

Because the value of the *condition* is evaluated after the *statements* are executed, the *statements* in a `repeat-while` statement are executed at least once.

The value of the *condition* must be of type `Bool` or a type bridged to `Bool`. The condition can also be an optional binding declaration, as discussed in Optional Binding.

```
GRAMMAR OF A REPEAT-WHILE STATEMENT
repeat-while-statement → repeat code-block while expression
```

Branch Statements

Branch statements allow the program to execute certain parts of code depending on the value of one or more conditions. The values of the conditions specified in a branch statement control how the program branches and, therefore, what block of code is executed. Swift has three branch statements: an `if` statement, a `guard` statement, and a `switch` statement.

Control flow in an `if` statement or a `switch` statement can be changed by a `break` statement and is discussed in Break Statement below.

```
GRAMMAR OF A BRANCH STATEMENT
branch-statement → if-statement
branch-statement → guard-statement
branch-statement → switch-statement
```

If Statement

An `if` statement is used for executing code based on the evaluation of one or more conditions.

There are two basic forms of an `if` statement. In each form, the opening and closing braces are required.

The first form allows code to be executed only when a condition is true and has the following form:

```
if condition {
    statements
}
```

The second form of an `if` statement provides an additional *else clause* (introduced by the `else` keyword) and is used for executing one part of code when the condition is true and another part of code when the same condition is false. When a single else clause is present, an `if` statement has the following form:

```
if condition {
    statements to execute if condition is true
} else {
    statements to execute if condition is false
}
```

The else clause of an `if` statement can contain another `if` statement to test more than one condition. An `if` statement chained together in this way has the following form:

```
if condition 1 {
    statements to execute if condition 1 is true
} else if condition 2 {
    statements to execute if condition 2 is true
} else {
    statements to execute if both conditions are false
}
```

The value of any condition in an `if` statement must be of type `Bool` or a type bridged to `Bool`. The condition can also be an optional binding declaration, as discussed in Optional Binding.

GRAMMAR OF AN IF STATEMENT

if-statement → **if** *condition-list* *code-block* *else-clause* *opt*
else-clause → **else** *code-block* | **else** *if-statement*

Guard Statement
A `guard` statement is used to transfer program control out of a scope if one or more conditions aren't met.

A `guard` statement has the following form:

```
guard condition else {
    statements
}
```

The value of any condition in a `guard` statement must be of type `Bool` or a type bridged to `Bool`. The condition can also be an optional binding declaration, as discussed in Optional Binding.

Any constants or variables assigned a value from an optional binding declaration in a `guard` statement condition can be used for the rest of the guard statement's enclosing scope.

The `else` clause of a `guard` statement is required, and must either call a function with the `Never` return type or transfer program control outside the guard statement's enclosing scope using one of the following statements:

- `return`
- `break`
- `continue`
- `throw`

Control transfer statements are discussed in Control Transfer Statements below. For more information on functions with the `Never` return type, see Functions that Never Return.

GRAMMAR OF A GUARD STATEMENT

guard-statement → **guard** *condition-list* **else** *code-block*

Switch Statement
A `switch` statement allows certain blocks of code to be executed depending on the value of a control expression.

A `switch` statement has the following form:

```
switch control expression {
case pattern 1:
    statements
case pattern 2 where condition:
    statements
case pattern 3 where condition,
     pattern 4 where condition:
    statements
default:
    statements
}
```

The *control expression* of the `switch` statement is evaluated and then compared with the patterns specified in each case. If a match is found, the program executes the *statements* listed within the scope of that case. The scope of each case can't be empty. As a result, you must include at least one statement following the colon (`:`) of each case label. Use a single `break` statement if you don't intend to execute any code in the body of a matched case.

The values of expressions your code can branch on are very flexible. For example, in addition to the values of scalar types, such as integers and characters, your code can branch on the values of any type, including floating-point numbers, strings, tuples, instances of custom classes, and optionals. The value of the *control expression* can even be matched to the value of a case in an enumeration and checked for inclusion in a specified range of values. For examples of how to use these various types of values in `switch` statements, see Switch in Control Flow.

A `switch` case can optionally contain a `where` clause after each pattern. A *where clause* is introduced by the `where` keyword followed by an expression, and is used to provide an additional condition before a pattern in a case is considered matched to the *control expression*. If a `where` clause is present, the *statements* within the relevant case are executed only if the value of the *control expression* matches one of the patterns of the case and the expression of the `where` clause evaluates to `true`. For example, a *control expression* matches the case in the example below only if it's a tuple that contains two elements of the same value, such as `(1, 1)`.

```
case let (x, y) where x == y:
```

As the above example shows, patterns in a case can also bind constants using the `let` keyword (they can also bind variables using the `var` keyword). These constants (or variables) can then be referenced in a corresponding `where` clause and throughout the rest of the code within the scope of the case. If the case contains multiple patterns that match the control expression, all of the patterns must contain the same constant or variable bindings, and each bound variable or constant must have the same type in all of the case's patterns.

A `switch` statement can also include a default case, introduced by the `default` keyword. The code within a default case is executed only if no other cases match the control expression. A `switch` statement can include only one default case, which must appear at the end of the `switch` statement.

Although the actual execution order of pattern-matching operations, and in particular the evaluation order of patterns in cases, is unspecified, pattern matching in a `switch` statement behaves as if the evaluation is performed in source order—that is, the order in which they appear in source code. As a result, if multiple cases contain patterns that evaluate to the same value, and thus can match the value of the control expression, the program executes only the code within the first matching case in source order.

Switch Statements Must Be Exhaustive

In Swift, every possible value of the control expression's type must match the value of at least one pattern of a case. When this simply isn't feasible (for example, when the control expression's type is `Int`), you can include a default case to satisfy the requirement.

Switching Over Future Enumeration Cases

A *nonfrozen enumeration* is a special kind of enumeration that may gain new enumeration cases in the future—even after you compile and ship an app. Switching over a nonfrozen enumeration requires extra consideration. When a library's authors mark an enumeration as nonfrozen, they reserve the right to add new enumeration cases, and any code that interacts with that enumeration *must* be able to handle those future cases without being recompiled. Code that's compiled in library evolution mode, code in the standard library, Swift overlays for Apple frameworks, and C and Objective-C code can declare nonfrozen enumerations. For information about frozen and nonfrozen enumerations, see frozen.

When switching over a nonfrozen enumeration value, you always need to include a default case, even if every case of the enumeration already has a corresponding switch case. You can apply the `@unknown` attribute to the default case, which indicates that the default case should match only enumeration cases that are added in the future. Swift produces a warning if the default case matches any enumeration case that's known at compiler time. This future warning informs you that the library author added a new case to the enumeration that doesn't have a corresponding switch case.

The following example switches over all three existing cases of the standard library's `Mirror.AncestorRepresentation` enumeration. If you add additional cases in the future, the compiler generates a warning to indicate that you need to update the switch statement to take the new cases into account.

```
let representation: Mirror.AncestorRepresentation = .generated
switch representation {
case .customized:
    print("Use the nearest ancestor's implementation.")
case .generated:
    print("Generate a default mirror for all ancestor classes.")
case .suppressed:
    print("Suppress the representation of all ancestor classes.")
@unknown default:
    print("Use a representation that was unknown when this code was compiled.")
}
// Prints "Generate a default mirror for all ancestor classes."
```

Execution Does Not Fall Through Cases Implicitly

After the code within a matched case has finished executing, the program exits from the `switch` statement. Program execution doesn't continue or "fall through" to the next case or default case. That said, if you want execution to continue from one case to the next, explicitly include a `fallthrough` statement, which simply consists of the `fallthrough` keyword, in the case from which you want execution to continue. For more information about the `fallthrough` statement, see Fallthrough Statement below.

GRAMMAR OF A SWITCH STATEMENT

switch-statement → **switch** *expression* **{** *switch-cases* *opt* **}**
switch-cases → *switch-case* *switch-cases* *opt*
switch-case → *case-label* *statements*
switch-case → *default-label* *statements*
switch-case → *conditional-switch-case*
case-label → *attributes* *opt* **case** *case-item-list* **:**
case-item-list → *pattern* *where-clause* *opt* | *pattern* *where-clause* *opt* **,** *case-item-list*
default-label → *attributes* *opt* **default** **:**
where-clause → **where** *where-expression*
where-expression → *expression*
conditional-switch-case → *switch-if-directive-clause* *switch-elseif-directive-clauses* *opt* *switch-else-directive-clause* *opt* *endif-directive*
switch-if-directive-clause → *if-directive* *compilation-condition* *switch-cases* *opt*
switch-elseif-directive-clauses → *elseif-directive-clause* *switch-elseif-directive-clauses* *opt*
switch-elseif-directive-clause → *elseif-directive* *compilation-condition* *switch-cases* *opt*
switch-else-directive-clause → *else-directive* *switch-cases* *opt*

Labeled Statement

You can prefix a loop statement, an `if` statement, a `switch` statement, or a `do` statement with a *statement label*, which consists of the name of the label followed immediately by a colon (:). Use statement labels with `break` and `continue` statements to be explicit about how you want to change control flow in a loop statement or a `switch` statement, as discussed in Break Statement and Continue Statement below.

The scope of a labeled statement is the entire statement following the statement label. You can nest labeled statements, but the name of each statement label must be unique.

For more information and to see examples of how to use statement labels, see Labeled Statements in Control Flow.

GRAMMAR OF A LABELED STATEMENT

labeled-statement → *statement-label* *loop-statement*
labeled-statement → *statement-label* *if-statement*
labeled-statement → *statement-label* *switch-statement*
labeled-statement → *statement-label* *do-statement*
statement-label → *label-name* **:**
label-name → *identifier*

Control Transfer Statements

Control transfer statements can change the order in which code in your program is executed by unconditionally transferring program control from one piece of code to another. Swift has five control transfer statements: a `break` statement, a `continue` statement, a `fallthrough` statement, a `return` statement, and a `throw` statement.

GRAMMAR OF A CONTROL TRANSFER STATEMENT

control-transfer-statement → *break-statement*
control-transfer-statement → *continue-statement*
control-transfer-statement → *fallthrough-statement*
control-transfer-statement → *return-statement*
control-transfer-statement → *throw-statement*

Break Statement

A `break` statement ends program execution of a loop, an `if` statement, or a `switch` statement. A `break` statement can consist of only the `break` keyword, or it can consist of the `break` keyword followed by the name of a statement label, as shown below.

```
break
break label name
```

When a `break` statement is followed by the name of a statement label, it ends program execution of the loop, `if` statement, or `switch` statement named by that label.

When a `break` statement isn't followed by the name of a statement label, it ends program execution of the `switch` statement or the innermost enclosing loop statement in which it occurs. You can't use an unlabeled `break` statement to break out of an `if` statement.

In both cases, program control is then transferred to the first line of code following the enclosing loop or `switch` statement, if any.

For examples of how to use a `break` statement, see Break and Labeled Statements in Control Flow.

GRAMMAR OF A BREAK STATEMENT

break-statement → **`break`** *label-name* *opt*

Continue Statement

A `continue` statement ends program execution of the current iteration of a loop statement but doesn't stop execution of the loop statement. A `continue` statement can consist of only the `continue` keyword, or it can consist of the `continue` keyword followed by the name of a statement label, as shown below.

```
continue
continue label name
```

When a `continue` statement is followed by the name of a statement label, it ends program execution of the current iteration of the loop statement named by that label.

When a `continue` statement isn't followed by the name of a statement label, it ends program execution of the current iteration of the innermost enclosing loop statement in which it occurs.

In both cases, program control is then transferred to the condition of the enclosing loop statement.

In a `for` statement, the increment expression is still evaluated after the `continue` statement is executed, because the increment expression is evaluated after the execution of the loop's body.

For examples of how to use a `continue` statement, see Continue and Labeled Statements in Control Flow.

GRAMMAR OF A CONTINUE STATEMENT

continue-statement → **`continue`** *label-name* *opt*

Fallthrough Statement

A `fallthrough` statement consists of the `fallthrough` keyword and occurs only in a case block of a `switch` statement. A `fallthrough` statement causes program execution to continue from one case in a `switch` statement to the next case. Program execution continues to the next case even if the patterns of the case label don't match the value of the `switch` statement's control expression.

A `fallthrough` statement can appear anywhere inside a `switch` statement, not just as the last statement of a case block, but it can't be used in the final case block. It also can't transfer control into a case block whose pattern contains value binding patterns.

For an example of how to use a `fallthrough` statement in a `switch` statement, see Control Transfer Statements in Control Flow.

> GRAMMAR OF A FALLTHROUGH STATEMENT
>
> *fallthrough-statement* → **`fallthrough`**

Return Statement

A `return` statement occurs in the body of a function or method definition and causes program execution to return to the calling function or method. Program execution continues at the point immediately following the function or method call.

A `return` statement can consist of only the `return` keyword, or it can consist of the `return` keyword followed by an expression, as shown below.

```
return
return expression
```

When a `return` statement is followed by an expression, the value of the expression is returned to the calling function or method. If the value of the expression doesn't match the value of the return type declared in the function or method declaration, the expression's value is converted to the return type before it's returned to the calling function or method.

> NOTE
>
> As described in Failable Initializers, a special form of the `return` statement (`return nil`) can be used in a failable initializer to indicate initialization failure.

When a `return` statement isn't followed by an expression, it can be used only to return from a function or method that doesn't return a value (that is, when the return type of the function or method is `Void` or `()`).

```
GRAMMAR OF A RETURN STATEMENT
return-statement → return expression opt
```

Throw Statement

A `throw` statement occurs in the body of a throwing function or method, or in the body of a closure expression whose type is marked with the `throws` keyword.

A `throw` statement causes a program to end execution of the current scope and begin error propagation to its enclosing scope. The error that's thrown continues to propagate until it's handled by a `catch` clause of a `do` statement.

A `throw` statement consists of the `throw` keyword followed by an expression, as shown below.

```
throw expression
```

The value of the *expression* must have a type that conforms to the `Error` protocol.

For an example of how to use a `throw` statement, see Propagating Errors Using Throwing Functions in Error Handling.

```
GRAMMAR OF A THROW STATEMENT
throw-statement → throw expression
```

Defer Statement

A `defer` statement is used for executing code just before transferring program control outside of the scope that the `defer` statement appears in.

A `defer` statement has the following form:

```
defer {
    statements
}
```

The statements within the `defer` statement are executed no matter how program control is transferred. This means that a `defer` statement can be used, for example, to perform manual resource management such as closing file descriptors, and to perform actions that need to happen even if an error is thrown.

If multiple `defer` statements appear in the same scope, the order they appear is the reverse of the order they're executed. Executing the last `defer` statement in a given scope first means that statements inside that last `defer` statement can refer to resources that will be cleaned up by other `defer` statements.

```
func f() {
    defer { print("First defer") }
    defer { print("Second defer") }
    print("End of function")
}
f()
// Prints "End of function"
// Prints "Second defer"
// Prints "First defer"
```

The statements in the `defer` statement can't transfer program control outside of the `defer` statement.

GRAMMAR OF A DEFER STATEMENT

defer-statement → **defer** *code-block*

Do Statement

The `do` statement is used to introduce a new scope and can optionally contain one or more `catch` clauses, which contain patterns that match against defined error conditions. Variables and constants declared in the scope of a `do` statement can be accessed only within that scope.

A `do` statement in Swift is similar to curly braces (`{}`) in C used to delimit a code block, and doesn't incur a performance cost at runtime.

A `do` statement has the following form:

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
} catch pattern 3, pattern 4 where condition {
    statements
} catch {
    statements
}
```

If any statement in the `do` code block throws an error, program control is transferred to the first `catch` clause whose pattern matches the error. If none of the clauses match, the error propagates to the surrounding scope. If an error is unhandled at the top level, program execution stops with a runtime error.

Like a `switch` statement, the compiler attempts to infer whether `catch` clauses are exhaustive. If such a determination can be made, the error is considered handled. Otherwise, the error can propagate out of the containing scope, which means the error must be handled by an enclosing `catch` clause or the containing function must be declared with `throws`.

A `catch` clause that has multiple patterns matches the error if any of its patterns match the error. If a `catch` clause contains multiple patterns, all of the patterns must contain the same constant or variable bindings, and each bound variable or constant must have the same type in all of the `catch` clause's patterns.

To ensure that an error is handled, use a `catch` clause with a pattern that matches all errors, such as a wildcard pattern (_). If a `catch` clause doesn't specify a pattern, the `catch` clause matches and binds any error to a local constant named `error`. For more information about the patterns you can use in a `catch` clause, see Patterns.

To see an example of how to use a `do` statement with several `catch` clauses, see Handling Errors.

GRAMMAR OF A DO STATEMENT

do-statement → **do** *code-block* *catch-clauses* *opt*
catch-clauses → *catch-clause* *catch-clauses* *opt*
catch-clause → **catch** *catch-pattern-list* *opt* *code-block*
catch-pattern-list → *catch-pattern* | *catch-pattern* **,** *catch-pattern-list*
catch-pattern → *pattern* *where-clause* *opt*

Compiler Control Statements

Compiler control statements allow the program to change aspects of the compiler's behavior. Swift has three compiler control statements: a conditional compilation block a line control statement, and a compile-time diagnostic statement.

GRAMMAR OF A COMPILER CONTROL STATEMENT

compiler-control-statement → *conditional-compilation-block*
compiler-control-statement → *line-control-statement*
compiler-control-statement → *diagnostic-statement*

Conditional Compilation Block

A conditional compilation block allows code to be conditionally compiled depending on the value of one or more compilation conditions.

Every conditional compilation block begins with the `#if` compilation directive and ends with the `#endif` compilation directive. A simple conditional compilation block has the following form:

```
#if compilation condition
statements
#endif
```

Unlike the condition of an `if` statement, the *compilation condition* is evaluated at compile time. As a result, the *statements* are compiled and executed only if the *compilation condition* evaluates to `true` at compile time.

The *compilation condition* can include the `true` and `false` Boolean literals, an identifier used with the `-D` command line flag, or any of the platform conditions listed in the table below.

Platform condition	Valid arguments
`os()`	`macOS`, `iOS`, `watchOS`, `tvOS`, `Linux`, `Windows`
`arch()`	`i386`, `x86_64`, `arm`, `arm64`
`swift()`	`>=` or `<` followed by a version number
`compiler()`	`>=` or `<` followed by a version number
`canImport()`	A module name
`targetEnvironment()`	`simulator`, `macCatalyst`

The version number for the `swift()` and `compiler()` platform conditions consists of a major number, optional minor number, optional patch number, and so on, with a dot (`.`) separating each part of the version number. There must not be whitespace between the comparison operator and the version number. The version for `compiler()` is the compiler version, regardless of the Swift version setting passed to the compiler. The version for `swift()` is the language version currently being compiled. For example, if you compile your code using the Swift 5 compiler in Swift 4.2 mode, the compiler

version is 5 and the language version is 4.2. With those settings, the following code prints all three messages:

```
#if compiler(>=5)
print("Compiled with the Swift 5 compiler or later")
#endif
#if swift(>=4.2)
print("Compiled in Swift 4.2 mode or later")
#endif
#if compiler(>=5) && swift(<5)
print("Compiled with the Swift 5 compiler or later in a Swift
     mode earlier than 5")
#endif
// Prints "Compiled with the Swift 5 compiler or later"
// Prints "Compiled in Swift 4.2 mode or later"
// Prints "Compiled with the Swift 5 compiler or later in a
     Swift mode earlier than 5"
```

The argument for the `canImport()` platform condition is the name of a module that may not be present on all platforms. This condition tests whether it's possible to import the module, but doesn't actually import it. If the module is present, the platform condition returns `true`; otherwise, it returns `false`.

The `targetEnvironment()` platform condition returns `true` when code is being compiled for the specified environment; otherwise, it returns `false`.

> NOTE
>
> The `arch(arm)` platform condition doesn't return `true` for ARM 64 devices. The `arch(i386)` platform condition returns `true` when code is compiled for the 32–bit iOS simulator.

You can combine and negate compilation conditions using the logical operators `&&`, `||`, and `!` and use parentheses for grouping. These operators have the same associativity and precedence as the logical operators that are used to combine ordinary Boolean expressions.

Similar to an `if` statement, you can add multiple conditional branches to test for different compilation conditions. You can add any number of additional branches using `#elseif` clauses. You can also add a final additional branch using an `#else` clause. Conditional compilation blocks that contain multiple branches have the following form:

```
#if compilation condition 1
statements to compile if compilation condition 1 is true
#elseif compilation condition 2
statements to compile if compilation condition 2 is true
#else
statements to compile if both compilation conditions are false

#endif
```

> NOTE
>
> Each statement in the body of a conditional compilation block is parsed even if it's not compiled. However, there's an exception if the compilation condition includes a `swift()` or `compiler()` platform condition: The statements are parsed only if the language or compiler version matches what is specified in the platform condition. This exception ensures that an older compiler doesn't attempt to parse syntax introduced in a newer version of Swift.

GRAMMAR OF A CONDITIONAL COMPILATION BLOCK

conditional-compilation-block → *if-directive-clause* *elseif-directive-clauses* *opt* *else-directive-clause* *opt* *endif-directive*
if-directive-clause → *if-directive* *compilation-condition* *statements* *opt*
elseif-directive-clauses → *elseif-directive-clause* *elseif-directive-clauses* *opt*
elseif-directive-clause → *elseif-directive* *compilation-condition* *statements* *opt*
else-directive-clause → *else-directive* *statements* *opt*
if-directive → **#if**
elseif-directive → **#elseif**
else-directive → **#else**
endif-directive → **#endif**
compilation-condition → *platform-condition*
compilation-condition → *identifier*
compilation-condition → *boolean-literal*
compilation-condition → **(** *compilation-condition* **)**
compilation-condition → **!** *compilation-condition*
compilation-condition → *compilation-condition* **&&** *compilation-condition*
compilation-condition → *compilation-condition* **||** *compilation-condition*
platform-condition → **os** **(** *operating-system* **)**
platform-condition → **arch** **(** *architecture* **)**
platform-condition → **swift** **(** **>=** *swift-version* **)** | **swift** **(** **<** *swift-version* **)**
platform-condition → **compiler** **(** **>=** *swift-version* **)** | **compiler** **(** **<** *swift-version* **)**
platform-condition → **canImport** **(** *module-name* **)**
platform-condition → **targetEnvironment** **(** *environment* **)**
operating-system → **macOS** | **iOS** | **watchOS** | **tvOS** | **Linux** | **Windows**
architecture → **i386** | **x86_64** | **arm** | **arm64**
swift-version → *decimal-digits* *swift-version-continuation* *opt*
swift-version-continuation → **.** *decimal-digits* *swift-version-continuation* *opt*
module-name → *identifier*
environment → **simulator** | **macCatalyst**

Line Control Statement

A line control statement is used to specify a line number and filename that can be different from the line number and filename of the source code being compiled. Use a line control statement to change the source code location used by Swift for diagnostic and debugging purposes.

A line control statement has the following forms:

```
#sourceLocation(file: file path, line: line number)
#sourceLocation()
```

The first form of a line control statement changes the values of the `#line`, `#file`, `#fileID`, and `#filePath` literal expressions, beginning with the line of code following the line control statement. The *line number* changes the value of `#line`, and is any integer literal greater than zero. The *file path* changes the value of `#file`, `#fileID`, and `#filePath`, and is a string literal. The specified string becomes the value of `#filePath`, and the last path component of the string is used by the value of `#fileID`. For information about `#file`, `#fileID`, and `#filePath`, see Literal Expression.

The second form of a line control statement, `#sourceLocation()`, resets the source code location back to the default line numbering and file path.

GRAMMAR OF A LINE CONTROL STATEMENT

line-control-statement → **#sourceLocation (file:** *file-path* **, line:** *line-number* **)**
line-control-statement → **#sourceLocation ( )**
line-number → A decimal integer greater than zero
file-path → *static-string-literal*

Compile-Time Diagnostic Statement
A compile-time diagnostic statement causes the compiler to emit an error or a warning during compilation. A compile-time diagnostic statement has the following forms:

```
#error(" error message ")
#warning(" warning message ")
```

The first form emits the *error message* as a fatal error and terminates the compilation process. The second form emits the *warning message* as a nonfatal warning and allows compilation to proceed. You write the diagnostic message as a static string literal. Static string literals can't use features like string interpolation or concatenation, but they can use the multiline string literal syntax.

GRAMMAR OF A COMPILE-TIME DIAGNOSTIC STATEMENT

diagnostic-statement → **#error (** *diagnostic-message* **)**
diagnostic-statement → **#warning (** *diagnostic-message* **)**
diagnostic-message → *static-string-literal*

Availability Condition

An *availability condition* is used as a condition of an `if`, `while`, and `guard` statement to query the availability of APIs at runtime, based on specified platforms arguments.

An availability condition has the following form:

```
if #available( platform name  version , ... , *) {
    statements to execute if the APIs are available
} else {

        fallback statements to execute if the APIs are unavailable

}
```

You use an availability condition to execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

The availability condition takes a comma-separated list of platform names and versions. Use `iOS`, `macOS`, `watchOS`, and `tvOS` for the platform names, and include the corresponding version numbers. The `*` argument is required and specifies that on any other platform, the body of the code block guarded by the availability condition executes on the minimum deployment target specified by your target.

Unlike Boolean conditions, you can't combine availability conditions using logical operators such as `&&` and `||`.

GRAMMAR OF AN AVAILABILITY CONDITION

availability-condition → **#available** **(** *availability-arguments* **)**
availability-arguments → *availability-argument* | *availability-argument* **,** *availability-arguments*
availability-argument → *platform-name* *platform-version*
availability-argument → *****
platform-name → **iOS** | **iOSApplicationExtension**
platform-name → **macOS** | **macOSApplicationExtension**
platform-name → **macCatalyst** | **macCatalystApplicationExtension**
platform-name → **watchOS**
platform-name → **tvOS**
platform-version → *decimal-digits*
platform-version → *decimal-digits* **.** *decimal-digits*
platform-version → *decimal-digits* **.** *decimal-digits* **.** *decimal-digits*

Declarations

A *declaration* introduces a new name or construct into your program. For example, you use declarations to introduce functions and methods, to introduce variables and constants, and to define enumeration, structure, class, and protocol types. You can also use a declaration to extend the behavior of an existing named type and to import symbols into your program that are declared elsewhere.

In Swift, most declarations are also definitions in the sense that they're implemented or initialized at the same time they're declared. That said, because protocols don't implement their members, most protocol members are declarations only. For convenience and because the distinction isn't that important in Swift, the term *declaration* covers both declarations and definitions.

GRAMMAR OF A DECLARATION

declaration → *import-declaration*
declaration → *constant-declaration*
declaration → *variable-declaration*
declaration → *typealias-declaration*
declaration → *function-declaration*
declaration → *enum-declaration*
declaration → *struct-declaration*
declaration → *class-declaration*
declaration → *actor-declaration*
declaration → *protocol-declaration*
declaration → *initializer-declaration*
declaration → *deinitializer-declaration*
declaration → *extension-declaration*
declaration → *subscript-declaration*
declaration → *operator-declaration*
declaration → *precedence-group-declaration*
declarations → *declaration* *declarations* $_{opt}$

Top-Level Code

The top-level code in a Swift source file consists of zero or more statements, declarations, and expressions. By default, variables, constants, and other named declarations that are declared at the top-level of a source file are accessible to code in every source file that's part of the same module. You can override this default behavior by marking the declaration with an access-level modifier, as described in Access Control Levels.

There are two kinds of top-level code: top-level declarations and executable top-level code. Top-level declarations consist of only declarations, and are allowed in all Swift source files. Executable top-level code contains statements and expressions, not just declarations, and is allowed only as the top-level entry point for the program.

The Swift code you compile to make an executable can contain at most one of the following approaches to mark the top-level entry point, regardless of how the code is organized into files and modules: the `main` attribute, the `NSApplicationMain` attribute, the `UIApplicationMain` attribute, a `main.swift` file, or a file that contains top-level executable code.

GRAMMAR OF A TOP-LEVEL DECLARATION

top-level-declaration → *statements* *opt*

Code Blocks

A *code block* is used by a variety of declarations and control structures to group statements together. It has the following form:

```
{
    statements
}
```

The *statements* inside a code block include declarations, expressions, and other kinds of statements and are executed in order of their appearance in source code.

GRAMMAR OF A CODE BLOCK

code-block → **{** *statements* $_{opt}$ **}**

Import Declaration

An *import declaration* lets you access symbols that are declared outside the current file. The basic form imports the entire module; it consists of the `import` keyword followed by a module name:

```
import module
```

Providing more detail limits which symbols are imported—you can specify a specific submodule or a specific declaration within a module or submodule. When this detailed form is used, only the imported symbol (and not the module that declares it) is made available in the current scope.

```
import import kind module.symbol name
import module.submodule
```

GRAMMAR OF AN IMPORT DECLARATION

import-declaration → *attributes* $_{opt}$ **import** *import-kind* $_{opt}$ *import-path*
import-kind → **typealias** | **struct** | **class** | **enum** | **protocol** | **let** | **var** | **func**
import-path → *import-path-identifier* | *import-path-identifier* **.** *import-path*
import-path-identifier → *identifier* | *operator*

Constant Declaration

A *constant declaration* introduces a constant named value into your program. Constant declarations are declared using the `let` keyword and have the following form:

```
let constant name: type = expression
```

A constant declaration defines an immutable binding between the *constant name* and the value of the initializer *expression*; after the value of a constant is set, it can't be changed. That said, if a constant is initialized with a class object, the object itself can change, but the binding between the constant name and the object it refers to can't.

When a constant is declared at global scope, it must be initialized with a value. When a constant declaration occurs in the context of a function or method, it can be initialized later, as long as it's guaranteed to have a value set before the first time its value is read. If the compiler can prove that the constant's value is never read, the constant isn't required to have a value set at all. When a constant declaration occurs in the context of a class or structure declaration, it's considered a *constant property*. Constant declarations aren't computed properties and therefore don't have getters or setters.

If the *constant name* of a constant declaration is a tuple pattern, the name of each item in the tuple is bound to the corresponding value in the initializer *expression*.

```
let (firstNumber, secondNumber) = (10, 42)
```

In this example, `firstNumber` is a named constant for the value `10`, and `secondNumber` is a named constant for the value `42`. Both constants can now be used independently:

```
print("The first number is \(firstNumber).")
// Prints "The first number is 10."
print("The second number is \(secondNumber).")
// Prints "The second number is 42."
```

The type annotation (`:` *type*) is optional in a constant declaration when the type of the *constant name* can be inferred, as described in Type Inference.

To declare a constant type property, mark the declaration with the `static` declaration modifier. A constant type property of a class is always implicitly final; you can't mark it with the `class` or `final` declaration modifier to allow or disallow overriding by subclasses. Type properties are discussed in Type Properties.

For more information about constants and for guidance about when to use them, see Constants and Variables and Stored Properties.

GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → *attributes* *opt* *declaration-modifiers* *opt* **let** *pattern-initializer-list*
pattern-initializer-list → *pattern-initializer* | *pattern-initializer* **,** *pattern-initializer-list*
pattern-initializer → *pattern* *initializer* *opt*
initializer → **=** *expression*

Variable Declaration

A *variable declaration* introduces a variable named value into your program and is declared using the `var` keyword.

Variable declarations have several forms that declare different kinds of named, mutable values, including stored and computed variables and

properties, stored variable and property observers, and static variable properties. The appropriate form to use depends on the scope at which the variable is declared and the kind of variable you intend to declare.

> NOTE
>
> You can also declare properties in the context of a protocol declaration, as described in Protocol Property Declaration.

You can override a property in a subclass by marking the subclass's property declaration with the `override` declaration modifier, as described in Overriding.

Stored Variables and Stored Variable Properties

The following form declares a stored variable or stored variable property:

```
var variable name: type = expression
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class or structure declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, it's referred to as a *stored variable*. When it's declared in the context of a class or structure declaration, it's referred to as a *stored variable property*.

The initializer *expression* can't be present in a protocol declaration, but in all other contexts, the initializer *expression* is optional. That said, if no initializer *expression* is present, the variable declaration must include an explicit type annotation (`:` *type*).

As with constant declarations, if the *variable name* is a tuple pattern, the name of each item in the tuple is bound to the corresponding value in the initializer *expression*.

As their names suggest, the value of a stored variable or a stored variable property is stored in memory.

Computed Variables and Computed Properties

The following form declares a computed variable or computed property:

```
var variable name: type {
    get {
        statements
    }
    set(setter name) {
        statements
    }
}
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class, structure, enumeration, or extension declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, it's referred to as a *computed variable*. When it's declared in the context of a class, structure, or extension declaration, it's referred to as a *computed property*.

The getter is used to read the value, and the setter is used to write the value. The setter clause is optional, and when only a getter is needed, you can omit both clauses and simply return the requested value directly, as described in Read-Only Computed Properties. But if you provide a setter clause, you must also provide a getter clause.

The *setter name* and enclosing parentheses is optional. If you provide a setter name, it's used as the name of the parameter to the setter. If you don't provide a setter name, the default parameter name to the setter is `newValue`, as described in Shorthand Setter Declaration.

Unlike stored named values and stored variable properties, the value of a computed named value or a computed property isn't stored in memory.

For more information and to see examples of computed properties, see Computed Properties.

Stored Variable Observers and Property Observers

You can also declare a stored variable or property with `willSet` and `didSet` observers. A stored variable or property declared with observers has the following form:

```
var variable name: type = expression {
    willSet(setter name) {
        statements
    }
    didSet(setter name) {
        statements
    }
}
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class or structure declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, the observers are referred to as *stored variable observers*. When it's declared in the context of a class or structure declaration, the observers are referred to as *property observers*.

You can add property observers to any stored property. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass, as described in Overriding Property Observers.

The initializer *expression* is optional in the context of a class or structure declaration, but required elsewhere. The *type* annotation is optional when the type can be inferred from the initializer *expression*. This expression is evaluated the first time you read the property's value. If you overwrite the property's initial value without reading it, this expression is evaluated before the first time you write to the property.

The `willSet` and `didSet` observers provide a way to observe (and to respond appropriately) when the value of a variable or property is being set. The observers aren't called when the variable or property is first initialized. Instead, they're called only when the value is set outside of an initialization context.

A `willSet` observer is called just before the value of the variable or property is set. The new value is passed to the `willSet` observer as a constant, and therefore it can't be changed in the implementation of the `willSet` clause. The `didSet` observer is called immediately after the new value is set. In contrast to the `willSet` observer, the old value of the variable or property is passed to the `didSet` observer in case you still need access to it. That said, if you assign a value to a variable or property within its own `didSet` observer clause, that new value that you assign will replace the one that was just set and passed to the `willSet` observer.

The *setter name* and enclosing parentheses in the `willSet` and `didSet` clauses are optional. If you provide setter names, they're used as the parameter names to the `willSet` and `didSet` observers. If you don't provide setter names, the default parameter name to the `willSet` observer is `newValue` and the default parameter name to the `didSet` observer is `oldValue`.

The `didSet` clause is optional when you provide a `willSet` clause. Likewise, the `willSet` clause is optional when you provide a `didSet` clause.

If the body of the `didSet` observer refers to the old value, the getter is called before the observer, to make the old value available. Otherwise, the

new value is stored without calling the superclass's getter. The example below shows a computed property that's defined by the superclass and overridden by its subclasses to add an observer.

```
class Superclass {
    private var xValue = 12
    var x: Int {
        get { print("Getter was called"); return xValue }
        set { print("Setter was called"); xValue = newValue }
    }
}

// This subclass doesn't refer to oldValue in its observer, so the
// superclass's getter is called only once to print the value.
class New: Superclass {
    override var x: Int {
        didSet { print("New value \(x)") }
    }
}
let new = New()
new.x = 100
// Prints "Setter was called"
// Prints "Getter was called"
// Prints "New value 100"
```

```
// This subclass refers to oldValue in its observer, so the superclass's
// getter is called once before the setter, and again to print the value.
class NewAndOld: Superclass {
    override var x: Int {
        didSet { print("Old value \(oldValue) - new value \(x)") }
    }
}
let newAndOld = NewAndOld()
newAndOld.x = 200
// Prints "Getter was called"
// Prints "Setter was called"
// Prints "Getter was called"
// Prints "Old value 12 - new value 200"
```

For more information and to see an example of how to use property observers, see Property Observers.

Type Variable Properties

To declare a type variable property, mark the declaration with the `static` declaration modifier. Classes can mark type computed properties with the `class` declaration modifier instead to allow subclasses to override the superclass's implementation. Type properties are discussed in Type Properties.

GRAMMAR OF A VARIABLE DECLARATION

variable-declaration → *variable-declaration-head* *pattern-initializer-list*
variable-declaration → *variable-declaration-head* *variable-name* *type-annotation* *code-block*
variable-declaration → *variable-declaration-head* *variable-name* *type-annotation* *getter-setter-block*
variable-declaration → *variable-declaration-head* *variable-name* *type-annotation* *getter-setter-keyword-block*
variable-declaration → *variable-declaration-head* *variable-name* *initializer* *willSet-didSet-block*
variable-declaration → *variable-declaration-head* *variable-name* *type-annotation* *initializer* *opt* *willSet-didSet-block*
variable-declaration-head → *attributes* *opt* *declaration-modifiers* *opt* **var**
variable-name → *identifier*
getter-setter-block → *code-block*
getter-setter-block → **{** *getter-clause* *setter-clause* *opt* **}**
getter-setter-block → **{** *setter-clause* *getter-clause* **}**
getter-clause → *attributes* *opt* *mutation-modifier* *opt* **get** *code-block*
setter-clause → *attributes* *opt* *mutation-modifier* *opt* **set** *setter-name* *opt* *code-block*
setter-name → **(** *identifier* **)**
getter-setter-keyword-block → **{** *getter-keyword-clause* *setter-keyword-clause* *opt* **}**
getter-setter-keyword-block → **{** *setter-keyword-clause* *getter-keyword-clause* **}**
getter-keyword-clause → *attributes* *opt* *mutation-modifier* *opt* **get**
setter-keyword-clause → *attributes* *opt* *mutation-modifier* *opt* **set**
willSet-didSet-block → **{** *willSet-clause* *didSet-clause* *opt* **}**
willSet-didSet-block → **{** *didSet-clause* *willSet-clause* *opt* **}**
willSet-clause → *attributes* *opt* **willSet** *setter-name* *opt* *code-block*
didSet-clause → *attributes* *opt* **didSet** *setter-name* *opt* *code-block*

Type Alias Declaration

A *type alias declaration* introduces a named alias of an existing type into your program. Type alias declarations are declared using the `typealias`

keyword and have the following form:

```
typealias name = existing type
```

After a type alias is declared, the aliased *name* can be used instead of the *existing type* everywhere in your program. The *existing type* can be a named type or a compound type. Type aliases don't create new types; they simply allow a name to refer to an existing type.

A type alias declaration can use generic parameters to give a name to an existing generic type. The type alias can provide concrete types for some or all of the generic parameters of the existing type. For example:

```
typealias StringDictionary<Value> =
     Dictionary<String, Value>

// The following dictionaries have the same type.
var dictionary1: StringDictionary<Int> = [:]
var dictionary2: Dictionary<String, Int> = [:]
```

When a type alias is declared with generic parameters, the constraints on those parameters must match exactly the constraints on the existing type's generic parameters. For example:

```
typealias DictionaryOfInts<Key: Hashable> =
     Dictionary<Key, Int>
```

Because the type alias and the existing type can be used interchangeably, the type alias can't introduce additional generic constraints.

A type alias can forward an existing type's generic parameters by omitting all generic parameters from the declaration. For example, the `Diccionario`

type alias declared here has the same generic parameters and constraints as `Dictionary`.

```
typealias Diccionario = Dictionary
```

Inside a protocol declaration, a type alias can give a shorter and more convenient name to a type that's used frequently. For example:

```
protocol Sequence {
    associatedtype Iterator: IteratorProtocol
    typealias Element = Iterator.Element
}

func sum<T: Sequence>(_ sequence: T) -> Int where
     T.Element == Int {
    // ...
}
```

Without this type alias, the `sum` function would have to refer to the associated type as `T.Iterator.Element` instead of `T.Element`.

See also Protocol Associated Type Declaration.

GRAMMAR OF A TYPE ALIAS DECLARATION

typealias-declaration → *attributes* *opt* *access-level-modifier* *opt*
typealias *typealias-name* *generic-parameter-clause* *opt* *typealias-assignment*
typealias-name → *identifier*
typealias-assignment → **=** *type*

Function Declaration

A *function declaration* introduces a function or method into your program. A function declared in the context of class, structure, enumeration, or protocol is referred to as a *method*. Function declarations are declared using the `func` keyword and have the following form:

```
func function name(parameters) -> return type {
    statements
}
```

If the function has a return type of `Void`, the return type can be omitted as follows:

```
func function name(parameters) {
    statements
}
```

The type of each parameter must be included—it can't be inferred. If you write `inout` in front of a parameter's type, the parameter can be modified inside the scope of the function. In-out parameters are discussed in detail in In-Out Parameters, below.

A function declaration whose *statements* include only a single expression is understood to return the value of that expression. This implicit return syntax is considered only when the expression's type and the function's return type aren't `Void` and aren't an enumeration like `Never` that doesn't have any cases.

Functions can return multiple values using a tuple type as the return type of the function.

A function definition can appear inside another function declaration. This kind of function is known as a *nested function*.

A nested function is nonescaping if it captures a value that's guaranteed to never escape—such as an in-out parameter—or passed as a nonescaping function argument. Otherwise, the nested function is an escaping function.

For a discussion of nested functions, see Nested Functions.

Parameter Names

Function parameters are a comma-separated list where each parameter has one of several forms. The order of arguments in a function call must match the order of parameters in the function's declaration. The simplest entry in a parameter list has the following form:

`parameter name`: `parameter type`

A parameter has a name, which is used within the function body, as well as an argument label, which is used when calling the function or method. By default, parameter names are also used as argument labels. For example:

```
1  func f(x: Int, y: Int) -> Int { return x + y }
2  f(x: 1, y: 2) // both x and y are labeled
```

You can override the default behavior for argument labels with one of the following forms:

`argument label` `parameter name`: `parameter type`
_ `parameter name`: `parameter type`

A name before the parameter name gives the parameter an explicit argument label, which can be different from the parameter name. The corresponding argument must use the given argument label in function or method calls.

An underscore (_) before a parameter name suppresses the argument label. The corresponding argument must have no label in function or method calls.

```
1  func repeatGreeting(_ greeting: String, count n:
       Int) { /* Greet n times */ }
2  repeatGreeting("Hello, world!", count: 2) //  count
       is labeled, greeting is not
```

In-Out Parameters

In-out parameters are passed as follows:

1. When the function is called, the value of the argument is copied.
2. In the body of the function, the copy is modified.
3. When the function returns, the copy's value is assigned to the original argument.

This behavior is known as *copy-in copy-out* or *call by value result*. For example, when a computed property or a property with observers is passed as an in-out parameter, its getter is called as part of the function call and its setter is called as part of the function return.

As an optimization, when the argument is a value stored at a physical address in memory, the same memory location is used both inside and outside the function body. The optimized behavior is known as *call by reference*; it satisfies all of the requirements of the copy-in copy-out model while removing the overhead of copying. Write your code using the model given by copy-in copy-out, without depending on the call-by-reference optimization, so that it behaves correctly with or without the optimization.

Within a function, don't access a value that was passed as an in-out argument, even if the original value is available in the current scope. Accessing the original is a simultaneous access of the value, which violates Swift's memory exclusivity guarantee. For the same reason, you can't pass the same value to multiple in-out parameters.

For more information about memory safety and memory exclusivity, see Memory Safety.

A closure or nested function that captures an in-out parameter must be nonescaping. If you need to capture an in-out parameter without mutating it, use a capture list to explicitly capture the parameter immutably.

```
func someFunction(a: inout Int) -> () -> Int {
    return { [a] in return a + 1 }
}
```

If you need to capture and mutate an in-out parameter, use an explicit local copy, such as in multithreaded code that ensures all mutation has finished before the function returns.

```
func multithreadedFunction(queue: DispatchQueue, x: inout Int) {
    // Make a local copy and manually copy it back.
    var localX = x
    defer { x = localX }

    // Operate on localX asynchronously, then wait before returning.
    queue.async { someMutatingOperation(&localX) }
    queue.sync {}
}
```

For more discussion and examples of in-out parameters, see [In-Out Parameters](#).

Special Kinds of Parameters

Parameters can be ignored, take a variable number of values, and provide default values using the following forms:

```
_ : parameter type
parameter name: parameter type...
parameter name: parameter type = default argument value
```

An underscore (_) parameter is explicitly ignored and can't be accessed within the body of the function.

A parameter with a base type name followed immediately by three dots (`...`) is understood as a variadic parameter. A parameter that immediately

follows a variadic parameter must have an argument label. A function can have multiple variadic parameters. A variadic parameter is treated as an array that contains elements of the base type name. For example, the variadic parameter `Int...` is treated as `[Int]`. For an example that uses a variadic parameter, see Variadic Parameters.

A parameter with an equals sign (=) and an expression after its type is understood to have a default value of the given expression. The given expression is evaluated when the function is called. If the parameter is omitted when calling the function, the default value is used instead.

```
func f(x: Int = 42) -> Int { return x }
f()        // Valid, uses default value
f(x: 7)    // Valid, uses the value provided
f(7)       // Invalid, missing argument label
```

Special Kinds of Methods

Methods on an enumeration or a structure that modify `self` must be marked with the `mutating` declaration modifier.

Methods that override a superclass method must be marked with the `override` declaration modifier. It's a compile-time error to override a method without the `override` modifier or to use the `override` modifier on a method that doesn't override a superclass method.

Methods associated with a type rather than an instance of a type must be marked with the `static` declaration modifier for enumerations and structures, or with either the `static` or `class` declaration modifier for classes. A class type method marked with the `class` declaration modifier can be overridden by a subclass implementation; a class type method marked with `class final` or `static` can't be overridden.

Methods with Special Names
Several methods that have special names enable syntactic sugar for function call syntax. If a type defines one of these methods, instances of the type can be used in function call syntax. The function call is understood to be a call to one of the specially named methods on that instance.

A class, structure, or enumeration type can support function call syntax by defining a `dynamicallyCall(withArguments:)` method or a `dynamicallyCall(withKeywordArguments:)` method, as described in dynamicCallable, or by defining a call-as-function method, as described below. If the type defines both a call-as-function method and one of the methods used by the `dynamicCallable` attribute, the compiler gives preference to the call-as-function method in circumstances where either method could be used.

The name of a call-as-function method is `callAsFunction()`, or another name that begins with `callAsFunction(` and adds labeled or unlabeled arguments—for example, `callAsFunction(_:_:)` and `callAsFunction(something:)` are also valid call-as-function method names.

The following function calls are equivalent:

```
struct CallableStruct {
    var value: Int
    func callAsFunction(_ number: Int, scale: Int) {
        print(scale * (number + value))
    }
}
let callable = CallableStruct(value: 100)
callable(4, scale: 2)
callable.callAsFunction(4, scale: 2)
// Both function calls print 208.
```

The call-as-function methods and the methods from the `dynamicCallable` attribute make different trade-offs between how much information you encode into the type system and how much dynamic behavior is possible at runtime. When you declare a call-as-function method, you specify the number of arguments, and each argument's type and label. The `dynamicCallable` attribute's methods specify only the type used to hold the array of arguments.

Defining a call-as-function method, or a method from the `dynamicCallable` attribute, doesn't let you use an instance of that type as if it were a function in any context other than a function call expression. For example:

```
let someFunction1: (Int, Int) -> Void =
     callable(_:scale:)  // Error
let someFunction2: (Int, Int) -> Void =
     callable.callAsFunction(_:scale:)
```

The `subscript(dynamicMember:)` subscript enables syntactic sugar for member lookup, as described in dynamicMemberLookup.

Throwing Functions and Methods

Functions and methods that can throw an error must be marked with the `throws` keyword. These functions and methods are known as *throwing functions* and *throwing methods*. They have the following form:

```
func function name(parameters) throws ->
      return type {
    statements
}
```

Calls to a throwing function or method must be wrapped in a `try` or `try!` expression (that is, in the scope of a `try` or `try!` operator).

The `throws` keyword is part of a function's type, and nonthrowing functions are subtypes of throwing functions. As a result, you can use a nonthrowing function in a context where as a throwing one is expected.

You can't overload a function based only on whether the function can throw an error. That said, you can overload a function based on whether a function *parameter* can throw an error.

A throwing method can't override a nonthrowing method, and a throwing method can't satisfy a protocol requirement for a nonthrowing method. That said, a nonthrowing method can override a throwing method, and a nonthrowing method can satisfy a protocol requirement for a throwing method.

Rethrowing Functions and Methods

A function or method can be declared with the `rethrows` keyword to indicate that it throws an error only if one of its function parameters throws an error. These functions and methods are known as *rethrowing functions* and *rethrowing methods*. Rethrowing functions and methods must have at least one throwing function parameter.

```
1  func someFunction(callback: () throws -> Void)
       rethrows {
2      try callback()
3  }
```

A rethrowing function or method can contain a `throw` statement only inside a `catch` clause. This lets you call the throwing function inside a `do-catch` statement and handle errors in the `catch` clause by throwing a different error. In addition, the `catch` clause must handle only errors thrown by one of the rethrowing function's throwing parameters. For example, the following is invalid because the `catch` clause would handle the error thrown by `alwaysThrows()`.

```
func alwaysThrows() throws {
    throw SomeError.error
}
func someFunction(callback: () throws -> Void)
     rethrows {
    do {
        try callback()
        try alwaysThrows()  // Invalid,
     alwaysThrows() isn't a throwing parameter
    } catch {
        throw AnotherError.error
    }
}
```

A throwing method can't override a rethrowing method, and a throwing method can't satisfy a protocol requirement for a rethrowing method. That said, a rethrowing method can override a throwing method, and a rethrowing method can satisfy a protocol requirement for a throwing method.

Asynchronous Functions and Methods

Functions and methods that run asynchronously must be marked with the `async` keyword. These functions and methods are known as *asynchronous functions* and *asynchronous methods*. They have the following form:

```
func function name(parameters) async ->
      return type {
    statements
}
```

Calls to an asynchronous function or method must be wrapped in an `await` expression—that is, they must be in the scope of an `await` operator.

The `async` keyword is part of the function's type, and synchronous functions are subtypes of asynchronous functions. As a result, you can use a synchronous function in a context where an asynchronous function is expected. For example, you can override an asynchronous method with a synchronous method, and a synchronous method can satisfy a protocol requirement that requires an asynchronous method.

Functions that Never Return

Swift defines a `Never` type, which indicates that a function or method doesn't return to its caller. Functions and methods with the `Never` return type are called *nonreturning*. Nonreturning functions and methods either cause an irrecoverable error or begin a sequence of work that continues indefinitely. This means that code that would otherwise run immediately after the call is never executed. Throwing and rethrowing functions can transfer program control to an appropriate `catch` block, even when they're nonreturning.

A nonreturning function or method can be called to conclude the `else` clause of a guard statement, as discussed in Guard Statement.

You can override a nonreturning method, but the new method must preserve its return type and nonreturning behavior.

GRAMMAR OF A FUNCTION DECLARATION

function-declaration → *function-head* *function-name* *generic-parameter-clause* *opt* *function-signature* *generic-where-clause* *opt* *function-body* *opt*
function-head → *attributes* *opt* *declaration-modifiers* *opt* **func**
function-name → *identifier* | *operator*
function-signature → *parameter-clause* **async** *opt* **throws** *opt* *function-result* *opt*
function-signature → *parameter-clause* **async** *opt* **rethrows** *function-result* *opt*
function-result → **->** *attributes* *opt* *type*
function-body → *code-block*
parameter-clause → **( )** | **(** *parameter-list* **)**
parameter-list → *parameter* | *parameter* **,** *parameter-list*
parameter → *external-parameter-name* *opt* *local-parameter-name* *type-annotation* *default-argument-clause* *opt*
parameter → *external-parameter-name* *opt* *local-parameter-name* *type-annotation*
parameter → *external-parameter-name* *opt* *local-parameter-name* *type-annotation* **...**
external-parameter-name → *identifier*
local-parameter-name → *identifier*
default-argument-clause → **=** *expression*

Enumeration Declaration

An *enumeration declaration* introduces a named enumeration type into your program.

Enumeration declarations have two basic forms and are declared using the `enum` keyword. The body of an enumeration declared using either form contains zero or more values—called *enumeration cases*—and any number of declarations, including computed properties, instance methods, type methods, initializers, type aliases, and even other enumeration, structure,

class, and actor declarations. Enumeration declarations can't contain deinitializer or protocol declarations.

Enumeration types can adopt any number of protocols, but can't inherit from classes, structures, or other enumerations.

Unlike classes and structures, enumeration types don't have an implicitly provided default initializer; all initializers must be declared explicitly. Initializers can delegate to other initializers in the enumeration, but the initialization process is complete only after an initializer assigns one of the enumeration cases to `self`.

Like structures but unlike classes, enumerations are value types; instances of an enumeration are copied when assigned to variables or constants, or when passed as arguments to a function call. For information about value types, see Structures and Enumerations Are Value Types.

You can extend the behavior of an enumeration type with an extension declaration, as discussed in Extension Declaration.

Enumerations with Cases of Any Type

The following form declares an enumeration type that contains enumeration cases of any type:

```
enum enumeration name: adopted protocols {
    case enumeration case 1
    case
      enumeration case 2(associated value types)
}
```

Enumerations declared in this form are sometimes called *discriminated unions* in other programming languages.

In this form, each case block consists of the `case` keyword followed by one or more enumeration cases, separated by commas. The name of each case must be unique. Each case can also specify that it stores values of a given type. These types are specified in the *associated value types* tuple, immediately following the name of the case.

Enumeration cases that store associated values can be used as functions that create instances of the enumeration with the specified associated values. And just like functions, you can get a reference to an enumeration case and apply it later in your code.

```
enum Number {
    case integer(Int)
    case real(Double)
}
let f = Number.integer
// f is a function of type (Int) -> Number

// Apply f to create an array of Number instances
     with integer values
let evenInts: [Number] = [0, 2, 4, 6].map(f)
```

For more information and to see examples of cases with associated value types, see Associated Values.

Enumerations with Indirection

Enumerations can have a recursive structure, that is, they can have cases with associated values that are instances of the enumeration type itself. However, instances of enumeration types have value semantics, which means they have a fixed layout in memory. To support recursion, the compiler must insert a layer of indirection.

To enable indirection for a particular enumeration case, mark it with the `indirect` declaration modifier. An indirect case must have an associated value.

```
enum Tree<T> {
    case empty
    indirect case node(value: T, left: Tree, right:
     Tree)
}
```

To enable indirection for all the cases of an enumeration that have an associated value, mark the entire enumeration with the `indirect` modifier —this is convenient when the enumeration contains many cases that would each need to be marked with the `indirect` modifier.

An enumeration that's marked with the `indirect` modifier can contain a mixture of cases that have associated values and cases those that don't. That said, it can't contain any cases that are also marked with the `indirect` modifier.

Enumerations with Cases of a Raw-Value Type

The following form declares an enumeration type that contains enumeration cases of the same basic type:

```
enum enumeration name: raw-value type,
       adopted protocols {
    case enumeration case 1 = raw value 1
    case enumeration case 2 = raw value 2
}
```

In this form, each case block consists of the `case` keyword, followed by one or more enumeration cases, separated by commas. Unlike the cases in the first form, each case has an underlying value, called a *raw value*, of the same basic type. The type of these values is specified in the *raw-value type* and must represent an integer, floating-point number, string, or single character. In particular, the *raw-value type* must conform to the `Equatable` protocol and one of the following protocols: `ExpressibleByIntegerLiteral` for integer literals, `ExpressibleByFloatLiteral` for floating-point literals, `ExpressibleByStringLiteral` for string literals that contain any number of characters, and `ExpressibleByUnicodeScalarLiteral` or `ExpressibleByExtendedGraphemeClusterLiteral` for string literals that contain only a single character. Each case must have a unique name and be assigned a unique raw value.

If the raw-value type is specified as `Int` and you don't assign a value to the cases explicitly, they're implicitly assigned the values `0`, `1`, `2`, and so on. Each unassigned case of type `Int` is implicitly assigned a raw value that's automatically incremented from the raw value of the previous case.

```
enum ExampleEnum: Int {
    case a, b, c = 5, d
}
```

In the above example, the raw value of `ExampleEnum.a` is `0` and the value of `ExampleEnum.b` is `1`. And because the value of `ExampleEnum.c` is explicitly set to `5`, the value of `ExampleEnum.d` is automatically incremented from `5` and is therefore `6`.

If the raw-value type is specified as `String` and you don't assign values to the cases explicitly, each unassigned case is implicitly assigned a string with the same text as the name of that case.

```swift
enum GamePlayMode: String {
    case cooperative, individual, competitive
}
```

In the above example, the raw value of `GamePlayMode.cooperative` is `"cooperative"`, the raw value of `GamePlayMode.individual` is `"individual"`, and the raw value of `GamePlayMode.competitive` is `"competitive"`.

Enumerations that have cases of a raw-value type implicitly conform to the `RawRepresentable` protocol, defined in the Swift standard library. As a result, they have a `rawValue` property and a failable initializer with the signature `init?(rawValue: RawValue)`. You can use the `rawValue` property to access the raw value of an enumeration case, as in `ExampleEnum.b.rawValue`. You can also use a raw value to find a corresponding case, if there is one, by calling the enumeration's failable initializer, as in `ExampleEnum(rawValue: 5)`, which returns an optional case. For more information and to see examples of cases with raw-value types, see Raw Values.

Accessing Enumeration Cases

To reference the case of an enumeration type, use dot (`.`) syntax, as in `EnumerationType.enumerationCase`. When the enumeration type can be inferred from context, you can omit it (the dot is still required), as described in Enumeration Syntax and Implicit Member Expression.

To check the values of enumeration cases, use a `switch` statement, as shown in Matching Enumeration Values with a Switch Statement. The enumeration type is pattern-matched against the enumeration case patterns in the case blocks of the `switch` statement, as described in Enumeration Case Pattern.

GRAMMAR OF AN ENUMERATION DECLARATION

enum-declaration → *attributes* opt *access-level-modifier* opt *union-style-enum*
enum-declaration → *attributes* opt *access-level-modifier* opt *raw-value-style-enum*
union-style-enum → **indirect** opt **enum** *enum-name* *generic-parameter-clause* opt *type-inheritance-clause* opt *generic-where-clause* opt **{** *union-style-enum-members* opt **}**
union-style-enum-members → *union-style-enum-member* *union-style-enum-members* opt
union-style-enum-member → *declaration* | *union-style-enum-case-clause* | *compiler-control-statement*
union-style-enum-case-clause → *attributes* opt **indirect** opt **case** *union-style-enum-case-list*
union-style-enum-case-list → *union-style-enum-case* | *union-style-enum-case* **,** *union-style-enum-case-list*
union-style-enum-case → *enum-case-name* *tuple-type* opt
enum-name → *identifier*
enum-case-name → *identifier*
raw-value-style-enum → **enum** *enum-name* *generic-parameter-clause* opt *type-inheritance-clause* *generic-where-clause* opt **{** *raw-value-style-enum-members* **}**
raw-value-style-enum-members → *raw-value-style-enum-member* *raw-value-style-enum-members* opt
raw-value-style-enum-member → *declaration* | *raw-value-style-enum-case-clause* | *compiler-control-statement*
raw-value-style-enum-case-clause → *attributes* opt **case** *raw-value-style-enum-case-list*
raw-value-style-enum-case-list → *raw-value-style-enum-case* | *raw-value-style-enum-case* **,** *raw-value-style-enum-case-list*
raw-value-style-enum-case → *enum-case-name* *raw-value-assignment* opt
raw-value-assignment → **=** *raw-value-literal*
raw-value-literal → *numeric-literal* | *static-string-literal* | *boolean-literal*

Structure Declaration

A *structure declaration* introduces a named structure type into your program. Structure declarations are declared using the `struct` keyword and have the following form:

```
struct structure name: adopted protocols {
    declarations
}
```

The body of a structure contains zero or more *declarations*. These *declarations* can include both stored and computed properties, type properties, instance methods, type methods, initializers, subscripts, type aliases, and even other structure, class, actor, and enumeration declarations. Structure declarations can't contain deinitializer or protocol declarations. For a discussion and several examples of structures that include various kinds of declarations, see Structures and Classes.

Structure types can adopt any number of protocols, but can't inherit from classes, enumerations, or other structures.

There are three ways to create an instance of a previously declared structure:

- Call one of the initializers declared within the structure, as described in Initializers.
- If no initializers are declared, call the structure's memberwise initializer, as described in Memberwise Initializers for Structure Types.
- If no initializers are declared, and all properties of the structure declaration were given initial values, call the structure's default initializer, as described in Default Initializers.

The process of initializing a structure's declared properties is described in Initialization.

Properties of a structure instance can be accessed using dot (`.`) syntax, as described in Accessing Properties.

Structures are value types; instances of a structure are copied when assigned to variables or constants, or when passed as arguments to a function call. For information about value types, see Structures and Enumerations Are Value Types.

You can extend the behavior of a structure type with an extension declaration, as discussed in Extension Declaration.

GRAMMAR OF A STRUCTURE DECLARATION

struct-declaration → *attributes* *opt* *access-level-modifier* *opt* **struct** *struct-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *struct-body*
struct-name → *identifier*
struct-body → **{** *struct-members* *opt* **}**
struct-members → *struct-member* *struct-members* *opt*
struct-member → *declaration* | *compiler-control-statement*

Class Declaration

A *class declaration* introduces a named class type into your program. Class declarations are declared using the `class` keyword and have the following form:

```
class class name: superclass, adopted protocols
    {
    declarations
}
```

The body of a class contains zero or more *declarations*. These *declarations* can include both stored and computed properties, instance methods, type methods, initializers, a single deinitializer, subscripts, type aliases, and even other class, structure, actor, and enumeration declarations. Class declarations can't contain protocol declarations. For a discussion and several examples of classes that include various kinds of declarations, see Structures and Classes.

A class type can inherit from only one parent class, its *superclass*, but can adopt any number of protocols. The *superclass* appears first after the *class name* and colon, followed by any *adopted protocols*. Generic classes can inherit from other generic and nongeneric classes, but a nongeneric class can inherit only from other nongeneric classes. When you write the name of a generic superclass class after the colon, you must include the full name of that generic class, including its generic parameter clause.

As discussed in Initializer Declaration, classes can have designated and convenience initializers. The designated initializer of a class must initialize all of the class's declared properties and it must do so before calling any of its superclass's designated initializers.

A class can override properties, methods, subscripts, and initializers of its superclass. Overridden properties, methods, subscripts, and designated initializers must be marked with the `override` declaration modifier.

To require that subclasses implement a superclass's initializer, mark the superclass's initializer with the `required` declaration modifier. The subclass's implementation of that initializer must also be marked with the `required` declaration modifier.

Although properties and methods declared in the *superclass* are inherited by the current class, designated initializers declared in the *superclass* are only inherited when the subclass meets the conditions described in Automatic Initializer Inheritance. Swift classes don't inherit from a universal base class.

There are two ways to create an instance of a previously declared class:

- Call one of the initializers declared within the class, as described in Initializers.
- If no initializers are declared, and all properties of the class declaration were given initial values, call the class's default initializer, as described in Default Initializers.

Access properties of a class instance with dot (`.`) syntax, as described in Accessing Properties.

Classes are reference types; instances of a class are referred to, rather than copied, when assigned to variables or constants, or when passed as arguments to a function call. For information about reference types, see Classes Are Reference Types.

You can extend the behavior of a class type with an extension declaration, as discussed in Extension Declaration.

GRAMMAR OF A CLASS DECLARATION

class-declaration → *attributes* *opt* *access-level-modifier* *opt* **final** *opt* **class** *class-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *class-body*
class-declaration → *attributes* *opt* **final** *access-level-modifier* *opt* **class** *class-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *class-body*
class-name → *identifier*
class-body → **{** *class-members* *opt* **}**
class-members → *class-member* *class-members* *opt*
class-member → *declaration* | *compiler-control-statement*

Actor Declaration

An *actor declaration* introduces a named actor type into your program. Actor declarations are declared using the `actor` keyword and have the

following form:

```
actor actor name: adopted protocols {
    declarations
}
```

The body of an actor contains zero or more *declarations*. These *declarations* can include both stored and computed properties, instance methods, type methods, initializers, a single deinitializer, subscripts, type aliases, and even other class, structure, and enumeration declarations. For a discussion and several examples of actors that include various kinds of declarations, see Actors.

Actor types can adopt any number of protocols, but can't inherit from classes, enumerations, structures, or other actors. However, an actor that is marked with the `@objc` attribute implicitly conforms to the `NSObjectProtocol` protocol and is exposed to the Objective-C runtime as a subtype of `NSObject`.

There are two ways to create an instance of a previously declared actor:

- Call one of the initializers declared within the actor, as described in Initializers.
- If no initializers are declared, and all properties of the actor declaration were given initial values, call the actor's default initializer, as described in Default Initializers.

By default, members of an actor are isolated to that actor. Code, such as the body of a method or the getter for a property, is executed on that actor. Code within the actor can interact with them synchronously because that code is already running on the same actor, but code outside the actor must mark them with `await` to indicate that this code is asynchronously running code on another actor. Key paths can't refer to isolated members of an actor.

Actor-isolated stored properties can be passed as in-out parameters to synchronous functions, but not to asynchronous functions.

Actors can also have nonisolated members, whose declarations are marked with the `nonisolated` keyword. A nonisolated member executes like code outside of the actor: It can't interact with any of the actor's isolated state, and callers don't mark it with `await` when using it.

Members of an actor can be marked with the `@objc` attribute only if they are nonisolated or asynchronous.

The process of initializing an actor's declared properties is described in Initialization.

Properties of a actor instance can be accessed using dot (`.`) syntax, as described in Accessing Properties.

Actors are reference types; instances of an actor are referred to, rather than copied, when assigned to variables or constants, or when passed as arguments to a function call. For information about reference types, see Classes Are Reference Types.

You can extend the behavior of a structure type with an extension declaration, as discussed in Extension Declaration.

GRAMMAR OF AN ACTOR DECLARATION

actor-declaration → *attributes* *opt* *access-level-modifier* *opt* **actor** *actor-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *actor-body*
actor-name → *identifier*
actor-body → **{** *actor-members* *opt* **}**
actor-members → *actor-member* *actor-members* *opt*
actor-member → *declaration* | *compiler-control-statement*

Protocol Declaration

A *protocol declaration* introduces a named protocol type into your program. Protocol declarations are declared at global scope using the `protocol` keyword and have the following form:

```
protocol protocol name: inherited protocols {
    protocol member declarations
}
```

The body of a protocol contains zero or more *protocol member declarations*, which describe the conformance requirements that any type adopting the protocol must fulfill. In particular, a protocol can declare that conforming types must implement certain properties, methods, initializers, and subscripts. Protocols can also declare special kinds of type aliases, called *associated types*, that can specify relationships among the various declarations of the protocol. Protocol declarations can't contain class, structure, enumeration, or other protocol declarations. The *protocol member declarations* are discussed in detail below.

Protocol types can inherit from any number of other protocols. When a protocol type inherits from other protocols, the set of requirements from those other protocols are aggregated, and any type that inherits from the current protocol must conform to all those requirements. For an example of how to use protocol inheritance, see Protocol Inheritance.

> NOTE
>
> You can also aggregate the conformance requirements of multiple protocols using protocol composition types, as described in Protocol Composition Type and Protocol Composition.

You can add protocol conformance to a previously declared type by adopting the protocol in an extension declaration of that type. In the extension, you must implement all of the adopted protocol's requirements.

If the type already implements all of the requirements, you can leave the body of the extension declaration empty.

By default, types that conform to a protocol must implement all properties, methods, and subscripts declared in the protocol. That said, you can mark these protocol member declarations with the `optional` declaration modifier to specify that their implementation by a conforming type is optional. The `optional` modifier can be applied only to members that are marked with the `objc` attribute, and only to members of protocols that are marked with the `objc` attribute. As a result, only class types can adopt and conform to a protocol that contains optional member requirements. For more information about how to use the `optional` declaration modifier and for guidance about how to access optional protocol members—for example, when you're not sure whether a conforming type implements them—see [Optional Protocol Requirements](#).

The cases of an enumeration can satisfy protocol requirements for type members. Specifically, an enumeration case without any associated values satisfies a protocol requirement for a get-only type variable of type `Self`, and an enumeration case with associated values satisfies a protocol requirement for a function that returns `Self` whose parameters and their argument labels match the case's associated values. For example:

```
protocol SomeProtocol {
    static var someValue: Self { get }
    static func someFunction(x: Int) -> Self
}
enum MyEnum: SomeProtocol {
    case someValue
    case someFunction(x: Int)
}
```

To restrict the adoption of a protocol to class types only, include the `AnyObject` protocol in the *inherited protocols* list after the colon. For example, the following protocol can be adopted only by class types:

```
protocol SomeProtocol: AnyObject {
    /* Protocol members go here */
}
```

Any protocol that inherits from a protocol that's marked with the `AnyObject` requirement can likewise be adopted only by class types.

> NOTE
>
> If a protocol is marked with the `objc` attribute, the `AnyObject` requirement is implicitly applied to that protocol; there's no need to mark the protocol with the `AnyObject` requirement explicitly.

Protocols are named types, and thus they can appear in all the same places in your code as other named types, as discussed in Protocols as Types. However, you can't construct an instance of a protocol, because protocols don't actually provide the implementations for the requirements they specify.

You can use protocols to declare which methods a delegate of a class or structure should implement, as described in Delegation.

GRAMMAR OF A PROTOCOL DECLARATION

protocol-declaration → *attributes* *opt* *access-level-modifier* *opt* **protocol** *protocol-name* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *protocol-body*
protocol-name → *identifier*
protocol-body → **{** *protocol-members* *opt* **}**
protocol-members → *protocol-member* *protocol-members* *opt*
protocol-member → *protocol-member-declaration* | *compiler-control-statement*
protocol-member-declaration → *protocol-property-declaration*
protocol-member-declaration → *protocol-method-declaration*
protocol-member-declaration → *protocol-initializer-declaration*
protocol-member-declaration → *protocol-subscript-declaration*
protocol-member-declaration → *protocol-associated-type-declaration*
protocol-member-declaration → *typealias-declaration*

Protocol Property Declaration

Protocols declare that conforming types must implement a property by including a *protocol property declaration* in the body of the protocol declaration. Protocol property declarations have a special form of a variable declaration:

```
var property name: type { get set }
```

As with other protocol member declarations, these property declarations declare only the getter and setter requirements for types that conform to the protocol. As a result, you don't implement the getter or setter directly in the protocol in which it's declared.

The getter and setter requirements can be satisfied by a conforming type in a variety of ways. If a property declaration includes both the `get` and `set` keywords, a conforming type can implement it with a stored variable property or a computed property that's both readable and writeable (that is, one that implements both a getter and a setter). However, that property declaration can't be implemented as a constant property or a read-only

computed property. If a property declaration includes only the `get` keyword, it can be implemented as any kind of property. For examples of conforming types that implement the property requirements of a protocol, see Property Requirements.

To declare a type property requirement in a protocol declaration, mark the property declaration with the `static` keyword. Structures and enumerations that conform to the protocol declare the property with the `static` keyword, and classes that conform to the protocol declare the property with either the `static` or `class` keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the same keyword as the type they extend uses. Extensions that provide a default implementation for a type property requirement use the `static` keyword.

See also Variable Declaration.

GRAMMAR OF A PROTOCOL PROPERTY DECLARATION

protocol-property-declaration → *variable-declaration-head* *variable-name* *type-annotation* *getter-setter-keyword-block*

Protocol Method Declaration

Protocols declare that conforming types must implement a method by including a protocol method declaration in the body of the protocol declaration. Protocol method declarations have the same form as function declarations, with two exceptions: They don't include a function body, and you can't provide any default parameter values as part of the function declaration. For examples of conforming types that implement the method requirements of a protocol, see Method Requirements.

To declare a class or static method requirement in a protocol declaration, mark the method declaration with the `static` declaration modifier. Structures and enumerations that conform to the protocol declare the method with the `static` keyword, and classes that conform to the protocol declare the method with either the `static` or `class` keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the

same keyword as the type they extend uses. Extensions that provide a default implementation for a type method requirement use the `static` keyword.

See also Function Declaration.

GRAMMAR OF A PROTOCOL METHOD DECLARATION

protocol-method-declaration → *function-head* *function-name* *generic-parameter-clause* *opt* *function-signature* *generic-where-clause* *opt*

Protocol Initializer Declaration

Protocols declare that conforming types must implement an initializer by including a protocol initializer declaration in the body of the protocol declaration. Protocol initializer declarations have the same form as initializer declarations, except they don't include the initializer's body.

A conforming type can satisfy a nonfailable protocol initializer requirement by implementing a nonfailable initializer or an `init!` failable initializer. A conforming type can satisfy a failable protocol initializer requirement by implementing any kind of initializer.

When a class implements an initializer to satisfy a protocol's initializer requirement, the initializer must be marked with the `required` declaration modifier if the class isn't already marked with the `final` declaration modifier.

See also Initializer Declaration.

GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION

protocol-initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause* **throws** *opt* *generic-where-clause* *opt*
protocol-initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause* **rethrows** *generic-where-clause* *opt*

Protocol Subscript Declaration
Protocols declare that conforming types must implement a subscript by including a protocol subscript declaration in the body of the protocol declaration. Protocol subscript declarations have a special form of a subscript declaration:

```
subscript (parameters) -> return type { get set }
```

Subscript declarations only declare the minimum getter and setter implementation requirements for types that conform to the protocol. If the subscript declaration includes both the `get` and `set` keywords, a conforming type must implement both a getter and a setter clause. If the subscript declaration includes only the `get` keyword, a conforming type must implement *at least* a getter clause and optionally can implement a setter clause.

To declare a static subscript requirement in a protocol declaration, mark the subscript declaration with the `static` declaration modifier. Structures and enumerations that conform to the protocol declare the subscript with the `static` keyword, and classes that conform to the protocol declare the subscript with either the `static` or `class` keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the same keyword as the type they extend uses. Extensions that provide a default implementation for a static subscript requirement use the `static` keyword.

See also Subscript Declaration.

GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION

protocol-subscript-declaration → *subscript-head* *subscript-result* *generic-where-clause* *opt* *getter-setter-keyword-block*

Protocol Associated Type Declaration
Protocols declare associated types using the `associatedtype` keyword. An associated type provides an alias for a type that's used as part of a

protocol's declaration. Associated types are similar to type parameters in generic parameter clauses, but they're associated with `Self` in the protocol in which they're declared. In that context, `Self` refers to the eventual type that conforms to the protocol. For more information and examples, see Associated Types.

You use a generic `where` clause in a protocol declaration to add constraints to an associated types inherited from another protocol, without redeclaring the associated types. For example, the declarations of `SubProtocol` below are equivalent:

```
protocol SomeProtocol {
    associatedtype SomeType
}

protocol SubProtocolA: SomeProtocol {
    // This syntax produces a warning.
    associatedtype SomeType: Equatable
}

// This syntax is preferred.
protocol SubProtocolB: SomeProtocol where SomeType:
     Equatable { }
```

See also Type Alias Declaration.

GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION

protocol-associated-type-declaration → *attributes* *opt* *access-level-modifier* *opt* **associatedtype** *typealias-name* *type-inheritance-clause* *opt* *typealias-assignment* *opt* *generic-where-clause* *opt*

Initializer Declaration

An *initializer declaration* introduces an initializer for a class, structure, or enumeration into your program. Initializer declarations are declared using the `init` keyword and have two basic forms.

Structure, enumeration, and class types can have any number of initializers, but the rules and associated behavior for class initializers are different. Unlike structures and enumerations, classes have two kinds of initializers: designated initializers and convenience initializers, as described in Initialization.

The following form declares initializers for structures, enumerations, and designated initializers of classes:

```
init(parameters) {
    statements
}
```

A designated initializer of a class initializes all of the class's properties directly. It can't call any other initializers of the same class, and if the class has a superclass, it must call one of the superclass's designated initializers. If the class inherits any properties from its superclass, one of the superclass's designated initializers must be called before any of these properties can be set or modified in the current class.

Designated initializers can be declared in the context of a class declaration only and therefore can't be added to a class using an extension declaration.

Initializers in structures and enumerations can call other declared initializers to delegate part or all of the initialization process.

To declare convenience initializers for a class, mark the initializer declaration with the `convenience` declaration modifier.

```
convenience init(parameters) {
    statements
}
```

Convenience initializers can delegate the initialization process to another convenience initializer or to one of the class's designated initializers. That said, the initialization processes must end with a call to a designated initializer that ultimately initializes the class's properties. Convenience initializers can't call a superclass's initializers.

You can mark designated and convenience initializers with the `required` declaration modifier to require that every subclass implement the initializer. A subclass's implementation of that initializer must also be marked with the `required` declaration modifier.

By default, initializers declared in a superclass aren't inherited by subclasses. That said, if a subclass initializes all of its stored properties with default values and doesn't define any initializers of its own, it inherits all of the superclass's initializers. If the subclass overrides all of the superclass's designated initializers, it inherits the superclass's convenience initializers.

As with methods, properties, and subscripts, you need to mark overridden designated initializers with the `override` declaration modifier.

> NOTE
>
> If you mark an initializer with the `required` declaration modifier, you don't also mark the initializer with the `override` modifier when you override the required initializer in a subclass.

Just like functions and methods, initializers can throw or rethrow errors. And just like functions and methods, you use the `throws` or `rethrows` keyword after an initializer's parameters to indicate the appropriate behavior.

To see examples of initializers in various type declarations, see Initialization.

Failable Initializers

A *failable initializer* is a type of initializer that produces an optional instance or an implicitly unwrapped optional instance of the type the initializer is declared on. As a result, a failable initializer can return `nil` to indicate that initialization failed.

To declare a failable initializer that produces an optional instance, append a question mark to the `init` keyword in the initializer declaration (`init?`). To declare a failable initializer that produces an implicitly unwrapped optional instance, append an exclamation point instead (`init!`). The example below shows an `init?` failable initializer that produces an optional instance of a structure.

```
struct SomeStruct {
    let property: String
    // produces an optional instance of 'SomeStruct'
    init?(input: String) {
        if input.isEmpty {
            // discard 'self' and return 'nil'
            return nil
        }
        property = input
    }
}
```

You call an `init?` failable initializer in the same way that you call a nonfailable initializer, except that you must deal with the optionality of the

result.

```
if let actualInstance = SomeStruct(input: "Hello") {
    // do something with the instance of 'SomeStruct'
} else {
    // initialization of 'SomeStruct' failed and the initializer returned 'nil'
}
```

A failable initializer can return `nil` at any point in the implementation of the initializer's body.

A failable initializer can delegate to any kind of initializer. A nonfailable initializer can delegate to another nonfailable initializer or to an `init!` failable initializer. A nonfailable initializer can delegate to an `init?` failable initializer by force-unwrapping the result of the superclass's initializer—for example, by writing `super.init()!`.

Initialization failure propagates through initializer delegation. Specifically, if a failable initializer delegates to an initializer that fails and returns `nil`, then the initializer that delegated also fails and implicitly returns `nil`. If a nonfailable initializer delegates to an `init!` failable initializer that fails and returns `nil`, then a runtime error is raised (as if you used the `!` operator to unwrap an optional that has a `nil` value).

A failable designated initializer can be overridden in a subclass by any kind of designated initializer. A nonfailable designated initializer can be overridden in a subclass by a nonfailable designated initializer only.

For more information and to see examples of failable initializers, see [Failable Initializers](#).

GRAMMAR OF AN INITIALIZER DECLARATION

initializer-declaration → *initializer-head* *generic-parameter-clause* *opt*
parameter-clause **throws** *opt* *generic-where-clause* *opt* *initializer-body*
initializer-declaration → *initializer-head* *generic-parameter-clause* *opt*
parameter-clause **rethrows** *generic-where-clause* *opt* *initializer-body*
initializer-head → *attributes* *opt* *declaration-modifiers* *opt* **init**
initializer-head → *attributes* *opt* *declaration-modifiers* *opt* **init ?**
initializer-head → *attributes* *opt* *declaration-modifiers* *opt* **init !**
initializer-body → *code-block*

Deinitializer Declaration

A *deinitializer declaration* declares a deinitializer for a class type. Deinitializers take no parameters and have the following form:

```
deinit {
    statements
}
```

A deinitializer is called automatically when there are no longer any references to a class object, just before the class object is deallocated. A deinitializer can be declared only in the body of a class declaration—but not in an extension of a class—and each class can have at most one.

A subclass inherits its superclass's deinitializer, which is implicitly called just before the subclass object is deallocated. The subclass object isn't deallocated until all deinitializers in its inheritance chain have finished executing.

Deinitializers aren't called directly.

For an example of how to use a deinitializer in a class declaration, see Deinitialization.

GRAMMAR OF A DEINITIALIZER DECLARATION

deinitializer-declaration → *attributes* $_{opt}$ **deinit** *code-block*

Extension Declaration

An *extension declaration* allows you to extend the behavior of existing types. Extension declarations are declared using the `extension` keyword and have the following form:

```
extension type name where requirements {
    declarations
}
```

The body of an extension declaration contains zero or more *declarations*. These *declarations* can include computed properties, computed type properties, instance methods, type methods, initializers, subscript declarations, and even class, structure, and enumeration declarations. Extension declarations can't contain deinitializer or protocol declarations, stored properties, property observers, or other extension declarations. Declarations in a protocol extension can't be marked `final`. For a discussion and several examples of extensions that include various kinds of declarations, see Extensions.

If the *type name* is a class, structure, or enumeration type, the extension extends that type. If the *type name* is a protocol type, the extension extends all types that conform to that protocol.

Extension declarations that extend a generic type or a protocol with associated types can include *requirements*. If an instance of the extended

type or of a type that conforms to the extended protocol satisfies the *requirements*, the instance gains the behavior specified in the declaration.

Extension declarations can contain initializer declarations. That said, if the type you're extending is defined in another module, an initializer declaration must delegate to an initializer already defined in that module to ensure members of that type are properly initialized.

Properties, methods, and initializers of an existing type can't be overridden in an extension of that type.

Extension declarations can add protocol conformance to an existing class, structure, or enumeration type by specifying *adopted protocols*:

```
extension type name: adopted protocols where
      requirements {
    declarations
}
```

Extension declarations can't add class inheritance to an existing class, and therefore you can specify only a list of protocols after the *type name* and colon.

Conditional Conformance

You can extend a generic type to conditionally conform to a protocol, so that instances of the type conform to the protocol only when certain requirements are met. You add conditional conformance to a protocol by including *requirements* in an extension declaration.

Overridden Requirements Aren't Used in Some Generic Contexts

In some generic contexts, types that get behavior from conditional conformance to a protocol don't always use the specialized

implementations of that protocol's requirements. To illustrate this behavior, the following example defines two protocols and a generic type that conditionally conforms to both protocols.

```
1  protocol Loggable {
2      func log()
3  }
4  extension Loggable {
5      func log() {
6          print(self)
7      }
8  }
9
10 protocol TitledLoggable: Loggable {
11     static var logTitle: String { get }
12 }
13 extension TitledLoggable {
14     func log() {
15         print("\(Self.logTitle): \(self)")
16     }
17 }
18
19 struct Pair<T>: CustomStringConvertible {
20     let first: T
21     let second: T
22     var description: String {
23         return "(\(first), \(second))"
24     }
25 }
26
```

```
extension Pair: Loggable where T: Loggable { }
extension Pair: TitledLoggable where T:
     TitledLoggable {
    static var logTitle: String {
        return "Pair of '\(T.logTitle)'"
    }
}

extension String: TitledLoggable {
    static var logTitle: String {
        return "String"
    }
}
```

The `Pair` structure conforms to `Loggable` and `TitledLoggable` whenever its generic type conforms to `Loggable` or `TitledLoggable`, respectively. In the example below, `oneAndTwo` is an instance of `Pair<String>`, which conforms to `TitledLoggable` because `String` conforms to `TitledLoggable`. When the `log()` method is called on `oneAndTwo` directly, the specialized version containing the title string is used.

```
let oneAndTwo = Pair(first: "one", second: "two")
oneAndTwo.log()
// Prints "Pair of 'String': (one, two)"
```

However, when `oneAndTwo` is used in a generic context or as an instance of the `Loggable` protocol, the specialized version isn't used. Swift picks which implementation of `log()` to call by consulting only the minimum

requirements that `Pair` needs to conform to `Loggable`. For this reason, the default implementation provided by the `Loggable` protocol is used instead.

```
func doSomething<T: Loggable>(with x: T) {
    x.log()
}
doSomething(with: oneAndTwo)
// Prints "(one, two)"
```

When `log()` is called on the instance that's passed to `doSomething(_:)`, the customized title is omitted from the logged string.

Protocol Conformance Must Not Be Redundant

A concrete type can conform to a particular protocol only once. Swift marks redundant protocol conformances as an error. You're likely to encounter this kind of error in two kinds of situations. The first situation is when you explicitly conform to the same protocol multiple times, but with different requirements. The second situation is when you implicitly inherit from the same protocol multiple times. These situations are discussed in the sections below.

Resolving Explicit Redundancy

Multiple extensions on a concrete type can't add conformance to the same protocol, even if the extensions' requirements are mutually exclusive. This restriction is demonstrated in the example below. Two extension declarations attempt to add conditional conformance to the `Serializable` protocol, one for for arrays with `Int` elements, and one for arrays with `String` elements.

```swift
protocol Serializable {
    func serialize() -> Any
}

extension Array: Serializable where Element == Int {
    func serialize() -> Any {
        // implementation
    }
}
extension Array: Serializable where Element == String {
    func serialize() -> Any {
        // implementation
    }
}
// Error: redundant conformance of 'Array<Element>' to protocol 'Serializable'
```

If you need to add conditional conformance based on multiple concrete types, create a new protocol that each type can conform to and use that protocol as the requirement when declaring conditional conformance.

```
protocol SerializableInArray { }
extension Int: SerializableInArray { }
extension String: SerializableInArray { }

extension Array: Serializable where Element:
     SerializableInArray {
    func serialize() -> Any {
        // implementation
    }
}
```

Resolving Implicit Redundancy

When a concrete type conditionally conforms to a protocol, that type implicitly conforms to any parent protocols with the same requirements.

If you need a type to conditionally conform to two protocols that inherit from a single parent, explicitly declare conformance to the parent protocol. This avoids implicitly conforming to the parent protocol twice with different requirements.

The following example explicitly declares the conditional conformance of `Array` to `Loggable` to avoid a conflict when declaring its conditional conformance to both `TitledLoggable` and the new `MarkedLoggable` protocol.

```
protocol MarkedLoggable: Loggable {
    func markAndLog()
}

extension MarkedLoggable {
    func markAndLog() {
        print("----------")
        log()
    }
}

extension Array: Loggable where Element: Loggable {
     }
extension Array: TitledLoggable where Element:
     TitledLoggable {
    static var logTitle: String {
        return "Array of '\(Element.logTitle)'"
    }
}
extension Array: MarkedLoggable where Element:
     MarkedLoggable { }
```

Without the extension to explicitly declare conditional conformance to `Loggable`, the other `Array` extensions would implicitly create these declarations, resulting in an error:

```
extension Array: Loggable where Element:
     TitledLoggable { }
extension Array: Loggable where Element:
     MarkedLoggable { }
// Error: redundant conformance of 'Array<Element>'
     to protocol 'Loggable'
```

GRAMMAR OF AN EXTENSION DECLARATION

extension-declaration → *attributes* *opt* *access-level-modifier* *opt* **extension** *type-identifier* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *extension-body*
extension-body → **{** *extension-members* *opt* **}**
extension-members → *extension-member* *extension-members* *opt*
extension-member → *declaration* | *compiler-control-statement*

Subscript Declaration

A *subscript* declaration allows you to add subscripting support for objects of a particular type and are typically used to provide a convenient syntax for accessing the elements in a collection, list, or sequence. Subscript declarations are declared using the `subscript` keyword and have the following form:

```
subscript (parameters) -> return type {
    get {
        statements
    }
    set(setter name) {
        statements
    }
}
```

Subscript declarations can appear only in the context of a class, structure, enumeration, extension, or protocol declaration.

The *parameters* specify one or more indexes used to access elements of the corresponding type in a subscript expression (for example, the `i` in the expression `object[i]`). Although the indexes used to access the elements can be of any type, each parameter must include a type annotation to specify the type of each index. The *return type* specifies the type of the element being accessed.

As with computed properties, subscript declarations support reading and writing the value of the accessed elements. The getter is used to read the value, and the setter is used to write the value. The setter clause is optional, and when only a getter is needed, you can omit both clauses and simply return the requested value directly. That said, if you provide a setter clause, you must also provide a getter clause.

The *setter name* and enclosing parentheses are optional. If you provide a setter name, it's used as the name of the parameter to the setter. If you don't provide a setter name, the default parameter name to the setter is `value`. The type of the parameter to the setter is the same as the *return type*.

You can overload a subscript declaration in the type in which it's declared, as long as the *parameters* or the *return type* differ from the one you're overloading. You can also override a subscript declaration inherited from a

superclass. When you do so, you must mark the overridden subscript declaration with the `override` declaration modifier.

Subscript parameters follow the same rules as function parameters, with two exceptions. By default, the parameters used in subscripting don't have argument labels, unlike functions, methods, and initializers. However, you can provide explicit argument labels using the same syntax that functions, methods, and initializers use. In addition, subscripts can't have in-out parameters. A subscript parameter can have a default value, using the syntax described in Special Kinds of Parameters.

You can also declare subscripts in the context of a protocol declaration, as described in Protocol Subscript Declaration.

For more information about subscripting and to see examples of subscript declarations, see Subscripts.

Type Subscript Declarations

To declare a subscript that's exposed by the type, rather than by instances of the type, mark the subscript declaration with the `static` declaration modifier. Classes can mark type computed properties with the `class` declaration modifier instead to allow subclasses to override the superclass's implementation. In a class declaration, the `static` keyword has the same effect as marking the declaration with both the `class` and `final` declaration modifiers.

GRAMMAR OF A SUBSCRIPT DECLARATION

subscript-declaration → *subscript-head* *subscript-result* *generic-where-clause* *opt* *code-block*
subscript-declaration → *subscript-head* *subscript-result* *generic-where-clause* *opt* *getter-setter-block*
subscript-declaration → *subscript-head* *subscript-result* *generic-where-clause* *opt* *getter-setter-keyword-block*
subscript-head → *attributes* *opt* *declaration-modifiers* *opt* **subscript** *generic-parameter-clause* *opt* *parameter-clause*
subscript-result → **->** *attributes* *opt* *type*

Operator Declaration

An *operator declaration* introduces a new infix, prefix, or postfix operator into your program and is declared using the `operator` keyword.

You can declare operators of three different fixities: infix, prefix, and postfix. The *fixity* of an operator specifies the relative position of an operator to its operands.

There are three basic forms of an operator declaration, one for each fixity. The fixity of the operator is specified by marking the operator declaration with the `infix`, `prefix`, or `postfix` declaration modifier before the `operator` keyword. In each form, the name of the operator can contain only the operator characters defined in Operators.

The following form declares a new infix operator:

```
infix operator operator name: precedence group
```

An *infix operator* is a binary operator that's written between its two operands, such as the familiar addition operator (`+`) in the expression `1 + 2`.

Infix operators can optionally specify a precedence group. If you omit the precedence group for an operator, Swift uses the default precedence group, `DefaultPrecedence`, which specifies a precedence just higher than `TernaryPrecedence`. For more information, see Precedence Group Declaration.

The following form declares a new prefix operator:

```
prefix operator operator name
```

A *prefix operator* is a unary operator that's written immediately before its operand, such as the prefix logical NOT operator (`!`) in the expression `!a`.

Prefix operators declarations don't specify a precedence level. Prefix operators are nonassociative.

The following form declares a new postfix operator:

```
postfix operator operator name
```

A *postfix operator* is a unary operator that's written immediately after its operand, such as the postfix forced-unwrap operator (`!`) in the expression `a!`.

As with prefix operators, postfix operator declarations don't specify a precedence level. Postfix operators are nonassociative.

After declaring a new operator, you implement it by declaring a static method that has the same name as the operator. The static method is a member of one of the types whose values the operator takes as an argument —for example, an operator that multiplies a `Double` by an `Int` is implemented as a static method on either the `Double` or `Int` structure. If you're implementing a prefix or postfix operator, you must also mark that method declaration with the corresponding `prefix` or `postfix` declaration

modifier. To see an example of how to create and implement a new operator, see Custom Operators.

GRAMMAR OF AN OPERATOR DECLARATION

operator-declaration → *prefix-operator-declaration* | *postfix-operator-declaration* | *infix-operator-declaration*
prefix-operator-declaration → **prefix operator** *operator*
postfix-operator-declaration → **postfix operator** *operator*
infix-operator-declaration → **infix operator** *operator* *infix-operator-group* *opt*
infix-operator-group → **:** *precedence-group-name*

Precedence Group Declaration

A *precedence group declaration* introduces a new grouping for infix operator precedence into your program. The precedence of an operator specifies how tightly the operator binds to its operands, in the absence of grouping parentheses.

A precedence group declaration has the following form:

```
precedencegroup precedence group name {
    higherThan: lower group names
    lowerThan: higher group names
    associativity: associativity
    assignment: assignment
}
```

The *lower group names* and *higher group names* lists specify the new precedence group's relation to existing precedence groups. The `lowerThan` precedence group attribute may only be used to refer to precedence groups declared outside of the current module. When two operators compete with

each other for their operands, such as in the expression `2 + 3 * 5`, the operator with the higher relative precedence binds more tightly to its operands.

> NOTE
>
> Precedence groups related to each other using *lower group names* and *higher group names* must fit into a single relational hierarchy, but they *don't* have to form a linear hierarchy. This means it's possible to have precedence groups with undefined relative precedence. Operators from those precedence groups can't be used next to each other without grouping parentheses.

Swift defines numerous precedence groups to go along with the operators provided by the standard library. For example, the addition (`+`) and subtraction (`-`) operators belong to the `AdditionPrecedence` group, and the multiplication (`*`) and division (`/`) operators belong to the `MultiplicationPrecedence` group. For a complete list of precedence groups provided by the Swift standard library, see Operator Declarations.

The *associativity* of an operator specifies how a sequence of operators with the same precedence level are grouped together in the absence of grouping parentheses. You specify the associativity of an operator by writing one of the context-sensitive keywords `left`, `right`, or `none`—if your omit the associativity, the default is `none`. Operators that are left-associative group left-to-right. For example, the subtraction operator (`-`) is left-associative, so the expression `4 - 5 - 6` is grouped as `(4 - 5) - 6` and evaluates to `-7`. Operators that are right-associative group right-to-left, and operators that are specified with an associativity of `none` don't associate at all. Nonassociative operators of the same precedence level can't appear adjacent to each to other. For example, the `<` operator has an associativity of `none`, which means `1 < 2 < 3` isn't a valid expression.

The *assignment* of a precedence group specifies the precedence of an operator when used in an operation that includes optional chaining. When set to `true`, an operator in the corresponding precedence group uses the same grouping rules during optional chaining as the assignment operators from the standard library. Otherwise, when set to `false` or omitted,

operators in the precedence group follows the same optional chaining rules as operators that don't perform assignment.

GRAMMAR OF A PRECEDENCE GROUP DECLARATION

precedence-group-declaration → **precedencegroup** *precedence-group-name* **{** *precedence-group-attributes* *opt* **}**
precedence-group-attributes → *precedence-group-attribute* *precedence-group-attributes* *opt*
precedence-group-attribute → *precedence-group-relation*
precedence-group-attribute → *precedence-group-assignment*
precedence-group-attribute → *precedence-group-associativity*
precedence-group-relation → **higherThan** **:** *precedence-group-names*
precedence-group-relation → **lowerThan** **:** *precedence-group-names*
precedence-group-assignment → **assignment** **:** *boolean-literal*
precedence-group-associativity → **associativity** **:** **left**
precedence-group-associativity → **associativity** **:** **right**
precedence-group-associativity → **associativity** **:** **none**
precedence-group-names → *precedence-group-name* | *precedence-group-name* **,** *precedence-group-names*
precedence-group-name → *identifier*

Declaration Modifiers

Declaration modifiers are keywords or context-sensitive keywords that modify the behavior or meaning of a declaration. You specify a declaration modifier by writing the appropriate keyword or context-sensitive keyword between a declaration's attributes (if any) and the keyword that introduces the declaration.

`class`

> Apply this modifier to a member of a class to indicate that the member is a member of the class itself, rather than a member of instances of the class. Members of a superclass that have this modifier and don't have the `final` modifier can be overridden by subclasses.

`dynamic`

Apply this modifier to any member of a class that can be represented by Objective-C. When you mark a member declaration with the `dynamic` modifier, access to that member is always dynamically dispatched using the Objective-C runtime. Access to that member is never inlined or devirtualized by the compiler.

Because declarations marked with the `dynamic` modifier are dispatched using the Objective-C runtime, they must be marked with the `objc` attribute.

`final`

Apply this modifier to a class or to a property, method, or subscript member of a class. It's applied to a class to indicate that the class can't be subclassed. It's applied to a property, method, or subscript of a class to indicate that a class member can't be overridden in any subclass. For an example of how to use the `final` attribute, see Preventing Overrides.

`lazy`

Apply this modifier to a stored variable property of a class or structure to indicate that the property's initial value is calculated and stored at most once, when the property is first accessed. For an example of how to use the `lazy` modifier, see Lazy Stored Properties.

`optional`

Apply this modifier to a protocol's property, method, or subscript members to indicate that a conforming type isn't required to implement those members.

You can apply the `optional` modifier only to protocols that are marked with the `objc` attribute. As a result, only class types can adopt and conform to a protocol that contains optional member requirements.

For more information about how to use the `optional` modifier and for guidance about how to access optional protocol members—for example, when you're not sure whether a conforming type implements them—see Optional Protocol Requirements.

`required`

Apply this modifier to a designated or convenience initializer of a class to indicate that every subclass must implement that initializer. The subclass's implementation of that initializer must also be marked with the `required` modifier.

`static`

Apply this modifier to a member of a structure, class, enumeration, or protocol to indicate that the member is a member of the type, rather than a member of instances of that type. In the scope of a class declaration, writing the `static` modifier on a member declaration has the same effect as writing the `class` and `final` modifiers on that member declaration. However, constant type properties of a class are an exception: `static` has its normal, nonclass meaning there because you can't write `class` or `final` on those declarations.

`unowned`

Apply this modifier to a stored variable, constant, or stored property to indicate that the variable or property has an unowned reference to the object stored as its value. If you try to access the variable or property after the object has been deallocated, a runtime error is raised. Like a weak reference, the type of the property or value must be a class type; unlike a weak reference, the type is non-optional. For an example and more information about the `unowned` modifier, see Unowned References.

`unowned(safe)`

An explicit spelling of `unowned`.

`unowned(unsafe)`

Apply this modifier to a stored variable, constant, or stored property to indicate that the variable or property has an unowned reference to the object stored as its value. If you try to access the variable or property after the object has been deallocated, you'll access the memory at the location where the object used to be, which is a memory-unsafe operation. Like a weak reference, the type of the property or value must be a class type; unlike a weak reference, the type is non-optional. For an example and more information about the `unowned` modifier, see Unowned References.

`weak`

Apply this modifier to a stored variable or stored variable property to indicate that the variable or property has a weak reference to the object stored as its value. The type of the variable or property must be an optional class type. If you access the variable or property after the object has been deallocated, its value is `nil`. For an example and more information about the `weak` modifier, see Weak References.

Access Control Levels

Swift provides five levels of access control: open, public, internal, file private, and private. You can mark a declaration with one of the access-level modifiers below to specify the declaration's access level. Access control is discussed in detail in Access Control.

`open`

Apply this modifier to a declaration to indicate the declaration can be accessed and subclassed by code in the same module as the declaration. Declarations marked with the `open` access-level modifier can also be accessed and subclassed by code in a module that imports the module that contains that declaration.

`public`

> Apply this modifier to a declaration to indicate the declaration can be accessed and subclassed by code in the same module as the declaration. Declarations marked with the `public` access-level modifier can also be accessed (but not subclassed) by code in a module that imports the module that contains that declaration.

`internal`

> Apply this modifier to a declaration to indicate the declaration can be accessed only by code in the same module as the declaration. By default, most declarations are implicitly marked with the `internal` access-level modifier.

`fileprivate`

> Apply this modifier to a declaration to indicate the declaration can be accessed only by code in the same source file as the declaration.

`private`

> Apply this modifier to a declaration to indicate the declaration can be accessed only by code within the declaration's immediate enclosing scope.

For the purpose of access control, extensions to the same type that are in the same file share an access-control scope. If the type they extend is also in the same file, they share the type's access-control scope. Private members declared in the type's declaration can be accessed from extensions, and private members declared in one extension can be accessed from other extensions and from the type's declaration.

Each access-level modifier above optionally accepts a single argument, which consists of the `set` keyword enclosed in parentheses (for example, `private(set)`). Use this form of an access-level modifier when you want to specify an access level for the setter of a variable or subscript that's less

than or equal to the access level of the variable or subscript itself, as discussed in Getters and Setters.

GRAMMAR OF A DECLARATION MODIFIER

declaration-modifier → **class** | **convenience** | **dynamic** | **final** | **infix** | **lazy** | **optional** | **override** | **postfix** | **prefix** | **required** | **static** | **unowned** | **unowned** **(** **safe** **)** | **unowned** **(** **unsafe** **)** | **weak**
declaration-modifier → *access-level-modifier*
declaration-modifier → *mutation-modifier*
declaration-modifier → *actor-isolation-modifier*
declaration-modifiers → *declaration-modifier* *declaration-modifiers* *opt*
access-level-modifier → **private** | **private** **(** **set** **)**
access-level-modifier → **fileprivate** | **fileprivate** **(** **set** **)**
access-level-modifier → **internal** | **internal** **(** **set** **)**
access-level-modifier → **public** | **public** **(** **set** **)**
access-level-modifier → **open** | **open** **(** **set** **)**
mutation-modifier → **mutating** | **nonmutating**
actor-isolation-modifier → **nonisolated**

Attributes

There are two kinds of attributes in Swift—those that apply to declarations and those that apply to types. An attribute provides additional information about the declaration or type. For example, the `discardableResult` attribute on a function declaration indicates that, although the function returns a value, the compiler shouldn't generate a warning if the return value is unused.

You specify an attribute by writing the `@` symbol followed by the attribute's name and any arguments that the attribute accepts:

```
@attribute name
@attribute name(attribute arguments)
```

Some declaration attributes accept arguments that specify more information about the attribute and how it applies to a particular declaration. These *attribute arguments* are enclosed in parentheses, and their format is defined by the attribute they belong to.

Declaration Attributes

You can apply a declaration attribute to declarations only.

available
Apply this attribute to indicate a declaration's life cycle relative to certain Swift language versions or certain platforms and operating system versions.

The `available` attribute always appears with a list of two or more comma-separated attribute arguments. These arguments begin with one of the following platform or language names:

- `iOS`
- `iOSApplicationExtension`
- `macOS`
- `macOSApplicationExtension`
- `macCatalyst`
- `macCatalystApplicationExtension`
- `watchOS`
- `watchOSApplicationExtension`
- `tvOS`
- `tvOSApplicationExtension`
- `swift`

You can also use an asterisk (`*`) to indicate the availability of the declaration on all of the platform names listed above. An `available` attribute that specifies availability using a Swift version number can't use the asterisk.

The remaining arguments can appear in any order and specify additional information about the declaration's life cycle, including important milestones.

- The `unavailable` argument indicates that the declaration isn't available on the specified platform. This argument can't be used when specifying Swift version availability.

- The `introduced` argument indicates the first version of the specified platform or language in which the declaration was introduced. It has the following form:

  `introduced: version number`

  The *version number* consists of one to three positive integers, separated by periods.

- The `deprecated` argument indicates the first version of the specified platform or language in which the declaration was deprecated. It has the following form:

  `deprecated: version number`

  The optional *version number* consists of one to three positive integers, separated by periods. Omitting the version number indicates that the declaration is currently deprecated, without giving any information about when the deprecation occurred. If you omit the version number, omit the colon (`:`) as well.

- The `obsoleted` argument indicates the first version of the specified platform or language in which the declaration was obsoleted. When a declaration is obsoleted, it's removed from the specified platform or language and can no longer be used. It has the following form:

  `obsoleted: version number`

  The *version number* consists of one to three positive integers, separated by periods.

- The `message` argument provides a textual message that the compiler displays when emitting a warning or error about the use of a deprecated or obsoleted declaration. It has the following form:

`message: message`

The *message* consists of a string literal.

- The `renamed` argument provides a textual message that indicates the new name for a declaration that's been renamed. The compiler displays the new name when emitting an error about the use of a renamed declaration. It has the following form:

  `renamed: new name`

  The *new name* consists of a string literal.

  You can apply the `available` attribute with the `renamed` and `unavailable` arguments to a type alias declaration, as shown below, to indicate that the name of a declaration changed between releases of a framework or library. This combination results in a compile-time error that the declaration has been renamed.

```
// First release
protocol MyProtocol {
    // protocol definition
}
```

```
// Subsequent release renames MyProtocol
protocol MyRenamedProtocol {
    // protocol definition
}

@available(*, unavailable, renamed:
     "MyRenamedProtocol")
typealias MyProtocol = MyRenamedProtocol
```

You can apply multiple `available` attributes on a single declaration to specify the declaration's availability on different platforms and different versions of Swift. The declaration that the `available` attribute applies to is ignored if the attribute specifies a platform or language version that doesn't match the current target. If you use multiple `available` attributes, the effective availability is the combination of the platform and Swift availabilities.

If an `available` attribute only specifies an `introduced` argument in addition to a platform or language name argument, you can use the following shorthand syntax instead:

```
@available(platform name version number, *)
@available(swift version number)
```

The shorthand syntax for `available` attributes concisely expresses availability for multiple platforms. Although the two forms are functionally equivalent, the shorthand form is preferred whenever possible.

```swift
@available(iOS 10.0, macOS 10.12, *)
class MyClass {
    // class definition
}
```

An `available` attribute that specifies availability using a Swift version number can't additionally specify a declaration's platform availability. Instead, use separate `available` attributes to specify a Swift version availability and one or more platform availabilities.

```swift
@available(swift 3.0.2)
@available(macOS 10.12, *)
struct MyStruct {
    // struct definition
}
```

discardableResult
Apply this attribute to a function or method declaration to suppress the compiler warning when the function or method that returns a value is called without using its result.

dynamicCallable
Apply this attribute to a class, structure, enumeration, or protocol to treat instances of the type as callable functions. The type must implement either a `dynamicallyCall(withArguments:)` method, a `dynamicallyCall(withKeywordArguments:)` method, or both.

You can call an instance of a dynamically callable type as if it's a function that takes any number of arguments.

```swift
@dynamicCallable
struct TelephoneExchange {
    func dynamicallyCall(withArguments phoneNumber: [Int]) {
        if phoneNumber == [4, 1, 1] {
            print("Get Swift help on forums.swift.org")
        } else {
            print("Unrecognized number")
        }
    }
}

let dial = TelephoneExchange()

// Use a dynamic method call.
dial(4, 1, 1)
// Prints "Get Swift help on forums.swift.org"

dial(8, 6, 7, 5, 3, 0, 9)
// Prints "Unrecognized number"

// Call the underlying method directly.
dial.dynamicallyCall(withArguments: [4, 1, 1])
```

The declaration of the `dynamicallyCall(withArguments:)` method must have a single parameter that conforms to the `ExpressibleByArrayLiteral` protocol—like `[Int]` in the example above. The return type can be any type.

You can include labels in a dynamic method call if you implement the `dynamicallyCall(withKeywordArguments:)` method.

```
@dynamicCallable
struct Repeater {
    func dynamicallyCall(withKeywordArguments pairs: KeyValuePairs<String, Int>) -> String {
        return pairs
            .map { label, count in
                repeatElement(label, count: count).joined(separator: " ")
            }
            .joined(separator: "\n")
    }
}

let repeatLabels = Repeater()
print(repeatLabels(a: 1, b: 2, c: 3, b: 2, a: 1))
// a
// b b
// c c c
// b b
// a
```

The declaration of the `dynamicallyCall(withKeywordArguments:)` method must have a single parameter that conforms to the `ExpressibleByDictionaryLiteral` protocol, and the return type can be any type. The parameter's `Key` must be `ExpressibleByStringLiteral`. The previous example uses `KeyValuePairs` as the parameter type so that callers can include duplicate parameter labels—`a` and `b` appear multiple times in the call to `repeat`.

If you implement both `dynamicallyCall` methods, `dynamicallyCall(withKeywordArguments:)` is called when the method call includes keyword arguments. In all other cases, `dynamicallyCall(withArguments:)` is called.

You can only call a dynamically callable instance with arguments and a return value that match the types you specify in one of your `dynamicallyCall` method implementations. The call in the following example doesn't compile because there isn't an implementation of `dynamicallyCall(withArguments:)` that takes `KeyValuePairs<String, String>`.

```
repeatLabels(a: "four") // Error
```

dynamicMemberLookup

Apply this attribute to a class, structure, enumeration, or protocol to enable members to be looked up by name at runtime. The type must implement a `subscript(dynamicMember:)` subscript.

In an explicit member expression, if there isn't a corresponding declaration for the named member, the expression is understood as a call to the type's `subscript(dynamicMember:)` subscript, passing information about the member as the argument. The subscript can accept a parameter that's either a key path or a member name; if you implement both subscripts, the subscript that takes key path argument is used.

An implementation of `subscript(dynamicMember:)` can accept key paths using an argument of type `KeyPath`, `WritableKeyPath`, or `ReferenceWritableKeyPath`. It can accept member names using an argument of a type that conforms to the `ExpressibleByStringLiteral` protocol—in most cases, `String`. The subscript's return type can be any type.

Dynamic member lookup by member name can be used to create a wrapper type around data that can't be type checked at compile time, such as when bridging data from other languages into Swift. For example:

```
@dynamicMemberLookup
struct DynamicStruct {
    let dictionary = ["someDynamicMember": 325,
                      "someOtherMember": 787]
    subscript(dynamicMember member: String) -> Int {
        return dictionary[member] ?? 1054
    }
}
let s = DynamicStruct()

// Use dynamic member lookup.
let dynamic = s.someDynamicMember
print(dynamic)
// Prints "325"

// Call the underlying subscript directly.
let equivalent = s[dynamicMember:
     "someDynamicMember"]
print(dynamic == equivalent)
// Prints "true"
```

Dynamic member lookup by key path can be used to implement a wrapper type in a way that supports compile-time type checking. For example:

```swift
struct Point { var x, y: Int }

@dynamicMemberLookup
struct PassthroughWrapper<Value> {
    var value: Value
    subscript<T>(dynamicMember member:
     KeyPath<Value, T>) -> T {
        get { return value[keyPath: member] }
    }
}

let point = Point(x: 381, y: 431)
let wrapper = PassthroughWrapper(value: point)
print(wrapper.x)
```

frozen

Apply this attribute to a structure or enumeration declaration to restrict the kinds of changes you can make to the type. This attribute is allowed only when compiling in library evolution mode. Future versions of the library can't change the declaration by adding, removing, or reordering an enumeration's cases or a structure's stored instance properties. These changes are allowed on nonfrozen types, but they break ABI compatibility for frozen types.

> NOTE
>
> When the compiler isn't in library evolution mode, all structures and enumerations are implicitly frozen, and this attribute is ignored.

In library evolution mode, code that interacts with members of nonfrozen structures and enumerations is compiled in a way that allows it to continue working without recompiling even if a future version of the library adds, removes, or reorders some of that type's members. The compiler makes this possible using techniques like looking up information at runtime and adding a layer of indirection. Marking a structure or enumeration as frozen gives up this flexibility to gain performance: Future versions of the library can make only limited changes to the type, but the compiler can make additional optimizations in code that interacts with the type's members.

Frozen types, the types of the stored properties of frozen structures, and the associated values of frozen enumeration cases must be public or marked with the `usableFromInline` attribute. The properties of a frozen structure can't have property observers, and expressions that provide the initial value for stored instance properties must follow the same restrictions as inlinable functions, as discussed in inlinable.

To enable library evolution mode on the command line, pass the `-enable-library-evolution` option to the Swift compiler. To enable it in Xcode, set the "Build Libraries for Distribution" build setting (`BUILD_LIBRARY_FOR_DISTRIBUTION`) to Yes, as described in Xcode Help.

A switch statement over a frozen enumeration doesn't require a `default` case, as discussed in Switching Over Future Enumeration Cases. Including a `default` or `@unknown default` case when switching over a frozen enumeration produces a warning because that code is never executed.

GKInspectable

Apply this attribute to expose a custom GameplayKit component property to the SpriteKit editor UI. Applying this attribute also implies the `objc` attribute.

inlinable

Apply this attribute to a function, method, computed property, subscript, convenience initializer, or deinitializer declaration to expose that declaration's implementation as part of the module's public interface. The compiler is allowed to replace calls to an inlinable symbol with a copy of the symbol's implementation at the call site.

Inlinable code can interact with `public` symbols declared in any module, and it can interact with `internal` symbols declared in the same module that are marked with the `usableFromInline` attribute. Inlinable code can't interact with `private` or `fileprivate` symbols.

This attribute can't be applied to declarations that are nested inside functions or to `fileprivate` or `private` declarations. Functions and closures that are defined inside an inlinable function are implicitly inlinable, even though they can't be marked with this attribute.

main

Apply this attribute to a structure, class, or enumeration declaration to indicate that it contains the top-level entry point for program flow. The type must provide a `main` type function that doesn't take any arguments and returns `Void`. For example:

```
@main
struct MyTopLevel {
    static func main() {
        // Top-level code goes here
    }
}
```

Another way to describe the requirements of the `main` attribute is that the type you write this attribute on must satisfy the same requirements as types that conform to the following hypothetical protocol:

```
protocol ProvidesMain {
    static func main() throws
}
```

The Swift code you compile to make an executable can contain at most one top-level entry point, as discussed in Top-Level Code.

nonobjc

Apply this attribute to a method, property, subscript, or initializer declaration to suppress an implicit `objc` attribute. The `nonobjc` attribute tells the compiler to make the declaration unavailable in Objective-C code, even though it's possible to represent it in Objective-C.

Applying this attribute to an extension has the same effect as applying it to every member of that extension that isn't explicitly marked with the `objc` attribute.

You use the `nonobjc` attribute to resolve circularity for bridging methods in a class marked with the `objc` attribute, and to allow overloading of methods and initializers in a class marked with the `objc` attribute.

A method marked with the `nonobjc` attribute can't override a method marked with the `objc` attribute. However, a method marked with the `objc` attribute can override a method marked with the `nonobjc` attribute. Similarly, a method marked with the `nonobjc` attribute can't satisfy a protocol requirement for a method marked with the `objc` attribute.

NSApplicationMain

Apply this attribute to a class to indicate that it's the application delegate. Using this attribute is equivalent to calling the `NSApplicationMain(_:_:)` function.

If you don't use this attribute, supply a `main.swift` file with code at the top level that calls the `NSApplicationMain(_:_:)` function as follows:

```
import AppKit
NSApplicationMain(CommandLine.argc,
      CommandLine.unsafeArgv)
```

The Swift code you compile to make an executable can contain at most one top-level entry point, as discussed in Top-Level Code.

NSCopying

Apply this attribute to a stored variable property of a class. This attribute causes the property's setter to be synthesized with a *copy* of the property's value—returned by the `copyWithZone(_:)` method—instead of the value of the property itself. The type of the property must conform to the `NSCopying` protocol.

The `NSCopying` attribute behaves in a way similar to the Objective-C `copy` property attribute.

NSManaged

Apply this attribute to an instance method or stored variable property of a class that inherits from `NSManagedObject` to indicate that Core Data dynamically provides its implementation at runtime, based on the associated entity description. For a property marked with the `NSManaged` attribute, Core Data also provides the storage at runtime. Applying this attribute also implies the `objc` attribute.

objc

Apply this attribute to any declaration that can be represented in Objective-C—for example, nonnested classes, protocols, nongeneric enumerations (constrained to integer raw-value types), properties and methods (including getters and setters) of classes, protocols and optional members of a protocol, initializers, and subscripts. The `objc` attribute tells the compiler that a declaration is available to use in Objective-C code.

Applying this attribute to an extension has the same effect as applying it to every member of that extension that isn't explicitly marked with the `nonobjc` attribute.

The compiler implicitly adds the `objc` attribute to subclasses of any class defined in Objective-C. However, the subclass must not be generic, and must not inherit from any generic classes. You can explicitly add the `objc` attribute to a subclass that meets these criteria, to specify its Objective-C name as discussed below. Protocols that are marked with the `objc` attribute can't inherit from protocols that aren't marked with this attribute.

The `objc` attribute is also implicitly added in the following cases:

- The declaration is an override in a subclass, and the superclass's declaration has the `objc` attribute.
- The declaration satisfies a requirement from a protocol that has the `objc` attribute.
- The declaration has the `IBAction`, `IBSegueAction`, `IBOutlet`, `IBDesignable`, `IBInspectable`, `NSManaged`, or `GKInspectable` attribute.

If you apply the `objc` attribute to an enumeration, each enumeration case is exposed to Objective-C code as the concatenation of the enumeration name and the case name. The first letter of the case name is capitalized. For example, a case named `venus` in a Swift `Planet` enumeration is exposed to Objective-C code as a case named `PlanetVenus`.

The `objc` attribute optionally accepts a single attribute argument, which consists of an identifier. The identifier specifies the name to be exposed to Objective-C for the entity that the `objc` attribute applies to. You can use this argument to name classes, enumerations, enumeration cases, protocols, methods, getters, setters, and initializers. If you specify the Objective-C name for a class, protocol, or enumeration, include a three-letter prefix on the name, as described in Conventions in Programming with Objective-C. The example below exposes the getter for the `enabled` property of the `ExampleClass` to Objective-C code as `isEnabled` rather than just as the name of the property itself.

```
class ExampleClass: NSObject {
    @objc var enabled: Bool {
        @objc(isEnabled) get {
            // Return the appropriate value
        }
    }
}
```

For more information, see Importing Swift into Objective-C.

> NOTE
>
> The argument to the `objc` attribute can also change the runtime name for that declaration. You use the runtime name when calling functions that interact with the Objective-C runtime, like `NSClassFromString`, and when specifying class names in an app's Info.plist file. If you specify a name by passing an argument, that name is used as the name in Objective-C code and as the runtime name. If you omit the argument, the name used in Objective-C code matches the name in Swift code, and the runtime name follows the normal Swift compiler convention of name mangling.

objcMembers

Apply this attribute to a class declaration, to implicitly apply the `objc` attribute to all Objective-C compatible members of the class, its extensions, its subclasses, and all of the extensions of its subclasses.

Most code should use the `objc` attribute instead, to expose only the declarations that are needed. If you need to expose many declarations, you can group them in an extension that has the `objc` attribute. The `objcMembers` attribute is a convenience for libraries that make heavy use of the introspection facilities of the Objective-C runtime. Applying the `objc` attribute when it isn't needed can increase your binary size and adversely affect performance.

propertyWrapper

Apply this attribute to a class, structure, or enumeration declaration to use that type as a property wrapper. When you apply this attribute to a type, you create a custom attribute with the same name as the type. Apply that new attribute to a property of a class, structure, or enumeration to wrap access to the property through an instance of the wrapper type; apply the attribute to a local stored variable declaration to wrap access to the variable the same way. Computed variables, global variables, and constants can't use property wrappers.

The wrapper must define a `wrappedValue` instance property. The *wrapped value* of the property is the value that the getter and setter for this property expose. In most cases, `wrappedValue` is a computed value, but it can be a stored value instead. The wrapper defines and manages any underlying storage needed by its wrapped value. The compiler synthesizes storage for the instance of the wrapper type by prefixing the name of the wrapped property with an underscore (_)—for example, the wrapper for `someProperty` is stored as `_someProperty`. The synthesized storage for the wrapper has an access control level of `private`.

A property that has a property wrapper can include `willSet` and `didSet` blocks, but it can't override the compiler-synthesized `get` or `set` blocks.

Swift provides two forms of syntactic sugar for initialization of a property wrapper. You can use assignment syntax in the definition of a wrapped value to pass the expression on the right-hand side of the assignment as the argument to the `wrappedValue` parameter of the property wrapper's initializer. You can also provide arguments to the attribute when you apply it to a property, and those arguments are passed to the property wrapper's initializer. For example, in the code below, `SomeStruct` calls each of the initializers that `SomeWrapper` defines.

```
@propertyWrapper
struct SomeWrapper {
    var wrappedValue: Int
    var someValue: Double
    init() {
        self.wrappedValue = 100
        self.someValue = 12.3
    }
    init(wrappedValue: Int) {
        self.wrappedValue = wrappedValue
        self.someValue = 45.6
    }
    init(wrappedValue value: Int, custom: Double) {
        self.wrappedValue = value
        self.someValue = custom
    }
}

struct SomeStruct {
    // Uses init()
    @SomeWrapper var a: Int

    // Uses init(wrappedValue:)
    @SomeWrapper var b = 10

    // Both use init(wrappedValue:custom:)
```

```
27      @SomeWrapper(custom: 98.7) var c = 30
28      @SomeWrapper(wrappedValue: 30, custom: 98.7) var
         d
29  }
```

The *projected value* for a wrapped property is a second value that a property wrapper can use to expose additional functionality. The author of a property wrapper type is responsible for determining the meaning of its projected value and defining the interface that the projected value exposes. To project a value from a property wrapper, define a `projectedValue` instance property on the wrapper type. The compiler synthesizes an identifier for the projected value by prefixing the name of the wrapped property with a dollar sign (`$`)—for example, the projected value for `someProperty` is `$someProperty`. The projected value has the same access control level as the original wrapped property.

```swift
@propertyWrapper
struct WrapperWithProjection {
    var wrappedValue: Int
    var projectedValue: SomeProjection {
        return SomeProjection(wrapper: self)
    }
}
struct SomeProjection {
    var wrapper: WrapperWithProjection
}

struct SomeStruct {
    @WrapperWithProjection var x = 123
}
let s = SomeStruct()
s.x           // Int value
s.$x          // SomeProjection value
s.$x.wrapper  // WrapperWithProjection value
```

resultBuilder

Apply this attribute to a class, structure, enumeration to use that type as a result builder. A *result builder* is a type that builds a nested data structure step by step. You use result builders to implement a domain-specific language (DSL) for creating nested data structures in a natural, declarative way. For an example of how to use the `resultBuilder` attribute, see [Result Builders](www.appsdissected.com).

Result-Building Methods

A result builder implements static methods described below. Because all of the result builder's functionality is exposed through static methods, you don't ever initialize an instance of that type. The `buildBlock(_:)` method is required; the other methods—which enable additional functionality in the DSL—are optional. The declaration of a result builder type doesn't actually have to include any protocol conformance.

The description of the static methods uses three types as placeholders. The type `Expression` is a placeholder for the type of the result builder's input, `Component` is a placeholder for the type of a partial result, and `FinalResult` is a placeholder for the type of the result that the result builder produces. You replace these types with the actual types that your result builder uses. If your result-building methods don't specify a type for `Expression` or `FinalResult`, they default to being the same as `Component`.

The result-building methods are as follows:

```
static func buildBlock(_ components: Component...) ->
Component
```

> Combines an array of partial results into a single partial result. A result builder must implement this method.

```
static func buildOptional(_ component: Component?) ->
Component
```

> Builds a partial result from a partial result that can be `nil`. Implement this method to support `if` statements that don't include an `else` clause.

```
static func buildEither(first: Component) -> Component
```

> Builds a partial result whose value varies depending on some condition. Implement both this method and `buildEither(second:)` to support `switch` statements and `if` statements that include an `else` clause.

`static func buildEither(second: Component) -> Component`

Builds a partial result whose value varies depending on some condition. Implement both this method and `buildEither(first:)` to support `switch` statements and `if` statements that include an `else` clause.

`static func buildArray(_ components: [Component]) -> Component`

Builds a partial result from an array of partial results. Implement this method to support `for` loops.

`static func buildExpression(_ expression: Expression) -> Component`

Builds a partial result from an expression. You can implement this method to perform preprocessing—for example, converting expressions to an internal type—or to provide additional information for type inference at use sites.

`static func buildFinalResult(_ component: Component) -> FinalResult`

Builds a final result from a partial result. You can implement this method as part of a result builder that uses a different type for partial and final results, or to perform other postprocessing on a result before returning it.

`static func buildLimitedAvailability(_ component: Component) -> Component`

Builds a partial result that propagates or erases type information outside a compiler-control statement that performs an availability check. You can use this to erase type information that varies between the conditional branches.

For example, the code below defines a simple result builder that builds an array of integers. This code defines `Compontent` and `Expression` as type aliases, to make it easier to match the examples below to the list of methods above.

```
@resultBuilder
struct ArrayBuilder {
    typealias Component = [Int]
    typealias Expression = Int
    static func buildExpression(_ element: Expression) -> Component {
        return [element]
    }
    static func buildOptional(_ component: Component?) -> Component {
        guard let component = component else { return [] }
        return component
    }
    static func buildEither(first component: Component) -> Component {
        return component
    }
    static func buildEither(second component: Component) -> Component {
        return component
    }
    static func buildArray(_ components: [Component]) -> Component {
        return Array(components.joined())
    }
```

```
21      static func buildBlock(_ components:
         Component...) -> Component {
22          return Array(components.joined())
23      }
24  }
```

Result Transformations

The following syntactic transformations are applied recursively to turn code that uses result-builder syntax into code that calls the static methods of the result builder type:

- If the result builder has a `buildExpression(_:)` method, each expression becomes a call to that method. This transformation is always first. For example, the following declarations are equivalent:

```
1  @ArrayBuilder var builderNumber: [Int] { 10 }
2  var manualNumber =
       ArrayBuilder.buildExpression(10)
```

- An assignment statement is transformed like an expression, but is understood to evaluate to `()`. You can define an overload of `buildExpression(_:)` that takes an argument of type `()` to handle assignments specifically.

- A branch statement that checks an availability condition becomes a call to the `buildLimitedAvailability(_:)` method. This transformation happens before the transformation into a call to `buildEither(first:)`, `buildEither(second:)`, or `buildOptional(_:)`. You use the `buildLimitedAvailability(_:)` method to erase type information that changes depending on which branch is taken. For example, the `buildEither(first:)` and

`buildEither(second:)` methods below use a generic type that captures type information about both branches.

```
protocol Drawable {
    func draw() -> String
}
struct Text: Drawable {
    var content: String
    init(_ content: String) { self.content = content }
    func draw() -> String { return content }
}
struct Line<D: Drawable>: Drawable {
    var elements: [D]
    func draw() -> String {
        return elements.map { $0.draw() }.joined(separator: "")
    }
}
struct DrawEither<First: Drawable, Second: Drawable>: Drawable {
    var content: Drawable
    func draw() -> String { return content.draw() }
}

@resultBuilder
struct DrawingBuilder {
```

```
22      static func buildBlock<D: Drawable>(_
         components: D...) -> Line<D> {
23          return Line(elements: components)
24      }
25      static func buildEither<First, Second>
         (first: First)
26          -> DrawEither<First, Second> {
27              return DrawEither(content: first)
28      }
29      static func buildEither<First, Second>
         (second: Second)
30          -> DrawEither<First, Second> {
31              return DrawEither(content: second)
32      }
33  }
```

However, this approach causes a problem in code that has availability checks:

```
@available(macOS 99, *)
struct FutureText: Drawable {
    var content: String
    init(_ content: String) { self.content = content }
    func draw() -> String { return content }
}
@DrawingBuilder var brokenDrawing: Drawable {
    if #available(macOS 99, *) {
        FutureText("Inside.future")  // Problem
    } else {
        Text("Inside.present")
    }
}
// The type of brokenDrawing is Line<DrawEither<Line<FutureText>, Line<Text>>>
```

In the code above, `FutureText` appears as part of the type of `brokenDrawing` because it's one of the types in the `DrawEither` generic type. This could cause your program to crash if `FutureText` isn't available at runtime, even in the case where that type is explicitly not being used.

To solve this problem, implement a `buildLimitedAvailability(_:)` method to erase type information. For example, the code below builds an `AnyDrawable` value from its availability check.

```swift
struct AnyDrawable: Drawable {
    var content: Drawable
    func draw() -> String { return content.draw() }
}
extension DrawingBuilder {
    static func buildLimitedAvailability(_ content: Drawable) -> AnyDrawable {
        return AnyDrawable(content: content)
    }
}

@DrawingBuilder var typeErasedDrawing: Drawable {
    if #available(macOS 99, *) {
        FutureText("Inside.future")
    } else {
        Text("Inside.present")
    }
}
// The type of typeErasedDrawing is Line<DrawEither<AnyDrawable, Line<Text>>>
```

- A branch statement becomes a series of nested calls to the `buildEither(first:)` and `buildEither(second:)` methods. The statements' conditions and cases are mapped onto the leaf nodes of a binary tree, and the statement becomes a nested call to the

`buildEither` methods following the path to that leaf node from the root node.

For example, if you write a switch statement that has three cases, the compiler uses a binary tree with three leaf nodes. Likewise, because the path from the root node to the second case is “second child” and then “first child”, that case becomes a nested call like `buildEither(first: buildEither(second: ... ))`. The following declarations are equivalent:

```
1  let someNumber = 19
2  @ArrayBuilder var builderConditional: [Int] {
3      if someNumber < 12 {
4          31
5      } else if someNumber == 19 {
6          32
7      } else {
8          33
9      }
10 }
11
12 var manualConditional: [Int]
13 if someNumber < 12 {
14     let partialResult =
        ArrayBuilder.buildExpression(31)
15     let outerPartialResult =
        ArrayBuilder.buildEither(first:
        partialResult)
16     manualConditional =
        ArrayBuilder.buildEither(first:
        outerPartialResult)
17 } else if someNumber == 19 {
18     let partialResult =
        ArrayBuilder.buildExpression(32)
19     let outerPartialResult =
        ArrayBuilder.buildEither(second:
```

```
        partialResult)
    manualConditional =
        ArrayBuilder.buildEither(first:
        outerPartialResult)
} else {
    let partialResult =
        ArrayBuilder.buildExpression(33)
    manualConditional =
        ArrayBuilder.buildEither(second:
        partialResult)
}
```

- A branch statement that might not produce a value, like an `if` statement without an `else` clause, becomes a call to `buildOptional(_:)`. If the `if` statement's condition is satisfied, its code block is transformed and passed as the argument; otherwise, `buildOptional(_:)` is called with `nil` as its argument. For example, the following declarations are equivalent:

```
@ArrayBuilder var builderOptional: [Int] {
    if (someNumber % 2) == 1 { 20 }
}

var partialResult: [Int]? = nil
if (someNumber % 2) == 1 {
    partialResult =
     ArrayBuilder.buildExpression(20)
}
var manualOptional =
     ArrayBuilder.buildOptional(partialResult)
```

- A code block or `do` statement becomes a call to the `buildBlock(_:)` method. Each of the statements inside of the block is transformed, one at a time, and they become the arguments to the `buildBlock(_:)` method. For example, the following declarations are equivalent:

```
@ArrayBuilder var builderBlock: [Int] {
    100
    200
    300
}

var manualBlock = ArrayBuilder.buildBlock(
    ArrayBuilder.buildExpression(100),
    ArrayBuilder.buildExpression(200),
    ArrayBuilder.buildExpression(300)
)
```

- A `for` loop becomes a temporary variable, a `for` loop, and call to the `buildArray(_:)` method. The new `for` loop iterates over the sequence and appends each partial result to that array. The temporary array is passed as the argument in the `buildArray(_:)` call. For example, the following declarations are equivalent:

```
@ArrayBuilder var builderArray: [Int] {
    for i in 5...7 {
        100 + i
    }
}

var temporary: [[Int]] = []
for i in 5...7 {
    let partialResult =
     ArrayBuilder.buildExpression(100 + i)
    temporary.append(partialResult)
}
let manualArray =
     ArrayBuilder.buildArray(temporary)
```

- If the result builder has a `buildFinalResult(_:)` method, the final result becomes a call to that method. This transformation is always last.

Although the transformation behavior is described in terms of temporary variables, using a result builder doesn't actually create any new declarations that are visible from the rest of your code.

You can't use `break`, `continue`, `defer`, `guard`, or `return` statements, `while` statements, or `do-catch` statements in the code that a result builder transforms.

The transformation process doesn't change declarations in the code, which lets you use temporary constants and variables to build up expressions piece by piece. It also doesn't change `throw` statements, compile-time diagnostic statements, or closures that contain a `return` statement.

Whenever possible, transformations are coalesced. For example, the expression `4 + 5 * 6` becomes `buildExpression(4 + 5 * 6)` rather multiple calls to that function. Likewise, nested branch statements become a single binary tree of calls to the `buildEither` methods.

Custom Result-Builder Attributes

Creating a result builder type creates a custom attribute with the same name. You can apply that attribute in the following places:

- On a function declaration, the result builder builds the body of the function.
- On a variable or subscript declaration that includes a getter, the result builder builds the body of the getter.
- On a parameter in a function declaration, the result builder builds the body of a closure that's passed as the corresponding argument.

Applying a result builder attribute doesn't impact ABI compatibility. Applying a result builder attribute to a parameter makes that attribute part of the function's interface, which can effect source compatibility.

requires_stored_property_inits
Apply this attribute to a class declaration to require all stored properties within the class to provide default values as part of their definitions. This attribute is inferred for any class that inherits from `NSManagedObject`.

testable
Apply this attribute to an `import` declaration to import that module with changes to its access control that simplify testing the module's code. Entities in the imported module that are marked with the `internal` access-level modifier are imported as if they were declared with the `public`

access-level modifier. Classes and class members that are marked with the `internal` or `public` access-level modifier are imported as if they were declared with the `open` access-level modifier. The imported module must be compiled with testing enabled.

UIApplicationMain

Apply this attribute to a class to indicate that it's the application delegate. Using this attribute is equivalent to calling the `UIApplicationMain` function and passing this class's name as the name of the delegate class.

If you don't use this attribute, supply a `main.swift` file with code at the top level that calls the `UIApplicationMain(_:_:_:_:)` function. For example, if your app uses a custom subclass of `UIApplication` as its principal class, call the `UIApplicationMain(_:_:_:_:)` function instead of using this attribute.

The Swift code you compile to make an executable can contain at most one top-level entry point, as discussed in Top-Level Code.

usableFromInline

Apply this attribute to a function, method, computed property, subscript, initializer, or deinitializer declaration to allow that symbol to be used in inlinable code that's defined in the same module as the declaration. The declaration must have the `internal` access-level modifier. A structure or class marked `usableFromInline` can use only types that are public or `usableFromInline` for its properties. An enumeration marked `usableFromInline` can use only types that are public or `usableFromInline` for the raw values and associated values of its cases.

Like the `public` access-level modifier, this attribute exposes the declaration as part of the module's public interface. Unlike `public`, the compiler doesn't allow declarations marked with `usableFromInline` to be referenced by name in code outside the module, even though the

declaration's symbol is exported. However, code outside the module might still be able to interact with the declaration's symbol by using runtime behavior.

Declarations marked with the `inlinable` attribute are implicitly usable from inlinable code. Although either `inlinable` or `usableFromInline` can be applied to `internal` declarations, applying both attributes is an error.

warn_unqualified_access

Apply this attribute to a top-level function, instance method, or class or static method to trigger warnings when that function or method is used without a preceding qualifier, such as a module name, type name, or instance variable or constant. Use this attribute to help discourage ambiguity between functions with the same name that are accessible from the same scope.

For example, the Swift standard library includes both a top-level `min(_:_:)` function and a `min()` method for sequences with comparable elements. The sequence method is declared with the `warn_unqualified_access` attribute to help reduce confusion when attempting to use one or the other from within a `Sequence` extension.

Declaration Attributes Used by Interface Builder

Interface Builder attributes are declaration attributes used by Interface Builder to synchronize with Xcode. Swift provides the following Interface Builder attributes: `IBAction`, `IBSegueAction`, `IBOutlet`, `IBDesignable`, and `IBInspectable`. These attributes are conceptually the same as their Objective-C counterparts.

You apply the `IBOutlet` and `IBInspectable` attributes to property declarations of a class. You apply the `IBAction` and `IBSegueAction` attribute to method declarations of a class and the `IBDesignable` attribute to class declarations.

Applying the `IBAction`, `IBSegueAction`, `IBOutlet`, `IBDesignable`, or `IBInspectable` attribute also implies the `objc` attribute.

Type Attributes

You can apply type attributes to types only.

autoclosure
Apply this attribute to delay the evaluation of an expression by automatically wrapping that expression in a closure with no arguments. You apply it to a parameter's type in a function or method declaration, for a parameter whose type is a function type that takes no arguments and that returns a value of the type of the expression. For an example of how to use the `autoclosure` attribute, see Autoclosures and Function Type.

convention
Apply this attribute to the type of a function to indicate its calling conventions.

The `convention` attribute always appears with one of the following arguments:

- The `swift` argument indicates a Swift function reference. This is the standard calling convention for function values in Swift.
- The `block` argument indicates an Objective-C compatible block reference. The function value is represented as a reference to the block object, which is an `id`-compatible Objective-C object that embeds its invocation function within the object. The invocation function uses the C calling convention.

- The `c` argument indicates a C function reference. The function value carries no context and uses the C calling convention.

With a few exceptions, a function of any calling convention can be used when a function any other calling convention is needed. A nongeneric global function, a local function that doesn't capture any local variables, or a closure that doesn't capture any local variables can be converted to the C calling convention. Other Swift functions can't be converted to the C calling convention. A function with the Objective-C block calling convention can't be converted to the C calling convention.

escaping
Apply this attribute to a parameter's type in a function or method declaration to indicate that the parameter's value can be stored for later execution. This means that the value is allowed to outlive the lifetime of the call. Function type parameters with the `escaping` type attribute require explicit use of `self.` for properties or methods. For an example of how to use the `escaping` attribute, see Escaping Closures.

Switch Case Attributes

You can apply switch case attributes to switch cases only.

unknown
Apply this attribute to a switch case to indicate that it isn't expected to be matched by any case of the enumeration that's known at the time the code is compiled. For an example of how to use the `unknown` attribute, see Switching Over Future Enumeration Cases.

GRAMMAR OF AN ATTRIBUTE

attribute → **@** *attribute-name* *attribute-argument-clause* *opt*
attribute-name → *identifier*
attribute-argument-clause → **(** *balanced-tokens* *opt* **)**
attributes → *attribute* *attributes* *opt*
balanced-tokens → *balanced-token* *balanced-tokens* *opt*
balanced-token → **(** *balanced-tokens* *opt* **)**
balanced-token → **[** *balanced-tokens* *opt* **]**
balanced-token → **{** *balanced-tokens* *opt* **}**
balanced-token → Any identifier, keyword, literal, or operator
balanced-token → Any punctuation except **(**, **)**, **[**, **]**, **{**, or **}**

Converted by Evan at Apps Dissected -www.appsdissected.com

Patterns

A *pattern* represents the structure of a single value or a composite value. For example, the structure of a tuple `(1, 2)` is a comma-separated list of two elements. Because patterns represent the structure of a value rather than any one particular value, you can match them with a variety of values. For instance, the pattern `(x, y)` matches the tuple `(1, 2)` and any other two-element tuple. In addition to matching a pattern with a value, you can extract part or all of a composite value and bind each part to a constant or variable name.

In Swift, there are two basic kinds of patterns: those that successfully match any kind of value, and those that may fail to match a specified value at runtime.

The first kind of pattern is used for destructuring values in simple variable, constant, and optional bindings. These include wildcard patterns, identifier patterns, and any value binding or tuple patterns containing them. You can specify a type annotation for these patterns to constrain them to match only values of a certain type.

The second kind of pattern is used for full pattern matching, where the values you're trying to match against may not be there at runtime. These include enumeration case patterns, optional patterns, expression patterns, and type-casting patterns. You use these patterns in a case label of a `switch` statement, a `catch` clause of a `do` statement, or in the case condition of an `if`, `while`, `guard`, or `for-in` statement.

GRAMMAR OF A PATTERN

pattern → *wildcard-pattern* *type-annotation* *opt*
pattern → *identifier-pattern* *type-annotation* *opt*
pattern → *value-binding-pattern*
pattern → *tuple-pattern* *type-annotation* *opt*
pattern → *enum-case-pattern*
pattern → *optional-pattern*
pattern → *type-casting-pattern*
pattern → *expression-pattern*

Wildcard Pattern

A *wildcard pattern* matches and ignores any value and consists of an underscore (_). Use a wildcard pattern when you don't care about the values being matched against. For example, the following code iterates through the closed range `1...3`, ignoring the current value of the range on each iteration of the loop:

```
for _ in 1...3 {
    // Do something three times.
}
```

GRAMMAR OF A WILDCARD PATTERN

wildcard-pattern → _

Identifier Pattern

An *identifier pattern* matches any value and binds the matched value to a variable or constant name. For example, in the following constant

declaration, `someValue` is an identifier pattern that matches the value `42` of type `Int`:

```
let someValue = 42
```

When the match succeeds, the value `42` is bound (assigned) to the constant name `someValue`.

When the pattern on the left-hand side of a variable or constant declaration is an identifier pattern, the identifier pattern is implicitly a subpattern of a value-binding pattern.

GRAMMAR OF AN IDENTIFIER PATTERN

identifier-pattern → *identifier*

Value-Binding Pattern

A *value-binding pattern* binds matched values to variable or constant names. Value-binding patterns that bind a matched value to the name of a constant begin with the `let` keyword; those that bind to the name of variable begin with the `var` keyword.

Identifiers patterns within a value-binding pattern bind new named variables or constants to their matching values. For example, you can decompose the elements of a tuple and bind the value of each element to a corresponding identifier pattern.

```
let point = (3, 2)
switch point {
    // Bind x and y to the elements of point.
case let (x, y):
    print("The point is at (\(x), \(y)).")
}
// Prints "The point is at (3, 2)."
```

In the example above, `let` distributes to each identifier pattern in the tuple pattern `(x, y)`. Because of this behavior, the `switch` cases `case let (x, y):` and `case (let x, let y):` match the same values.

GRAMMAR OF A VALUE-BINDING PATTERN

value-binding-pattern → **var** *pattern* | **let** *pattern*

Tuple Pattern

A *tuple pattern* is a comma-separated list of zero or more patterns, enclosed in parentheses. Tuple patterns match values of corresponding tuple types.

You can constrain a tuple pattern to match certain kinds of tuple types by using type annotations. For example, the tuple pattern `(x, y): (Int, Int)` in the constant declaration `let (x, y): (Int, Int) = (1, 2)` matches only tuple types in which both elements are of type `Int`.

When a tuple pattern is used as the pattern in a `for-in` statement or in a variable or constant declaration, it can contain only wildcard patterns, identifier patterns, optional patterns, or other tuple patterns that contain those. For example, the following code isn't valid because the element `0` in the tuple pattern `(x, 0)` is an expression pattern:

```
let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2,
     1)]
// This code isn't valid.
for (x, 0) in points {
    /* ... */
}
```

The parentheses around a tuple pattern that contains a single element have no effect. The pattern matches values of that single element's type. For example, the following are equivalent:

```
let a = 2        // a: Int = 2
let (a) = 2      // a: Int = 2
let (a): Int = 2 // a: Int = 2
```

GRAMMAR OF A TUPLE PATTERN

tuple-pattern → (*tuple-pattern-element-list* *opt*)
tuple-pattern-element-list → *tuple-pattern-element* | *tuple-pattern-element* , *tuple-pattern-element-list*
tuple-pattern-element → *pattern* | *identifier* : *pattern*

Enumeration Case Pattern

An *enumeration case pattern* matches a case of an existing enumeration type. Enumeration case patterns appear in `switch` statement case labels and in the case conditions of `if`, `while`, `guard`, and `for-in` statements.

If the enumeration case you're trying to match has any associated values, the corresponding enumeration case pattern must specify a tuple pattern that

contains one element for each associated value. For an example that uses a `switch` statement to match enumeration cases containing associated values, see Associated Values.

An enumeration case pattern also matches values of that case wrapped in an optional. This simplified syntax lets you omit an optional pattern. Note that, because `Optional` is implemented as an enumeration, `.none` and `.some` can appear in the same switch as the cases of the enumeration type.

```
enum SomeEnum { case left, right }
let x: SomeEnum? = .left
switch x {
case .left:
    print("Turn left")
case .right:
    print("Turn right")
case nil:
    print("Keep going straight")
}
// Prints "Turn left"
```

GRAMMAR OF AN ENUMERATION CASE PATTERN

enum-case-pattern → *type-identifier* $_{opt}$. *enum-case-name* *tuple-pattern* $_{opt}$

Optional Pattern

An *optional pattern* matches values wrapped in a `some(Wrapped)` case of an `Optional<Wrapped>` enumeration. Optional patterns consist of an identifier pattern followed immediately by a question mark and appear in the same places as enumeration case patterns.

Because optional patterns are syntactic sugar for `Optional` enumeration case patterns, the following are equivalent:

```
let someOptional: Int? = 42
// Match using an enumeration case pattern.
if case .some(let x) = someOptional {
    print(x)
}

// Match using an optional pattern.
if case let x? = someOptional {
    print(x)
}
```

The optional pattern provides a convenient way to iterate over an array of optional values in a `for-in` statement, executing the body of the loop only for non-`nil` elements.

```
let arrayOfOptionalInts: [Int?] = [nil, 2, 3, nil,
     5]
// Match only non-nil values.
for case let number? in arrayOfOptionalInts {
    print("Found a \(number)")
}
// Found a 2
// Found a 3
// Found a 5
```

GRAMMAR OF AN OPTIONAL PATTERN

optional-pattern → *identifier-pattern* **?**

Type-Casting Patterns

There are two type-casting patterns, the `is` pattern and the `as` pattern. The `is` pattern appears only in `switch` statement case labels. The `is` and `as` patterns have the following form:

```
is type
pattern as type
```

The `is` pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the `is` pattern—or a subclass of that type. The `is` pattern behaves like the `is` operator in that they both perform a type cast but discard the returned type.

The `as` pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the `as` pattern—or a subclass of that type. If the match succeeds, the type of the matched value is cast to the *pattern* specified in the right-hand side of the `as` pattern.

For an example that uses a `switch` statement to match values with `is` and `as` patterns, see Type Casting for Any and AnyObject.

GRAMMAR OF A TYPE CASTING PATTERN

type-casting-pattern → *is-pattern* | *as-pattern*
is-pattern → **is** *type*
as-pattern → *pattern* **as** *type*

Expression Pattern

An *expression pattern* represents the value of an expression. Expression patterns appear only in `switch` statement case labels.

The expression represented by the expression pattern is compared with the value of an input expression using the Swift standard library `~=` operator. The matches succeeds if the `~=` operator returns `true`. By default, the `~=` operator compares two values of the same type using the `==` operator. It can also match a value with a range of values, by checking whether the value is contained within the range, as the following example shows.

```swift
let point = (1, 2)
switch point {
case (0, 0):
    print("(0, 0) is at the origin.")
case (-2...2, -2...2):
    print("(\(point.0), \(point.1)) is near the origin.")
default:
    print("The point is at (\(point.0), \(point.1)).")
}
// Prints "(1, 2) is near the origin."
```

You can overload the `~=` operator to provide custom expression matching behavior. For example, you can rewrite the above example to compare the `point` expression with a string representations of points.

```
// Overload the ~= operator to match a string with an integer.
func ~= (pattern: String, value: Int) -> Bool {
    return pattern == "\(value)"
}
switch point {
case ("0", "0"):
    print("(0, 0) is at the origin.")
default:
    print("The point is at (\(point.0), \(point.1)).")
}
// Prints "The point is at (1, 2)."
```

GRAMMAR OF AN EXPRESSION PATTERN

expression-pattern → *expression*

Generic Parameters and Arguments

This chapter describes parameters and arguments for generic types, functions, and initializers. When you declare a generic type, function, subscript, or initializer, you specify the type parameters that the generic type, function, or initializer can work with. These type parameters act as placeholders that are replaced by actual concrete type arguments when an instance of a generic type is created or a generic function or initializer is called.

For an overview of generics in Swift, see Generics.

Generic Parameter Clause

A *generic parameter clause* specifies the type parameters of a generic type or function, along with any associated constraints and requirements on those parameters. A generic parameter clause is enclosed in angle brackets (<>) and has the following form:

```
<generic parameter list>
```

The *generic parameter list* is a comma-separated list of generic parameters, each of which has the following form:

```
type parameter : constraint
```

A generic parameter consists of a *type parameter* followed by an optional *constraint*. A *type parameter* is simply the name of a placeholder type (for example, `T`, `U`, `V`, `Key`, `Value`, and so on). You have access to the type parameters (and any of their associated types) in the rest of the type,

function, or initializer declaration, including in the signature of the function or initializer.

The *constraint* specifies that a type parameter inherits from a specific class or conforms to a protocol or protocol composition. For example, in the generic function below, the generic parameter `T: Comparable` indicates that any type argument substituted for the type parameter `T` must conform to the `Comparable` protocol.

```
func simpleMax<T: Comparable>(_ x: T, _ y: T) -> T {
    if x < y {
        return y
    }
    return x
}
```

Because `Int` and `Double`, for example, both conform to the `Comparable` protocol, this function accepts arguments of either type. In contrast with generic types, you don't specify a generic argument clause when you use a generic function or initializer. The type arguments are instead inferred from the type of the arguments passed to the function or initializer.

```
simpleMax(17, 42) // T is inferred to be Int
simpleMax(3.14159, 2.71828) // T is inferred to be
     Double
```

Generic Where Clauses

You can specify additional requirements on type parameters and their associated types by including a generic `where` clause right before the opening curly brace of a type or function's body. A generic `where` clause

consists of the `where` keyword, followed by a comma-separated list of one or more *requirements*.

```
where requirements
```

The *requirements* in a generic `where` clause specify that a type parameter inherits from a class or conforms to a protocol or protocol composition. Although the generic `where` clause provides syntactic sugar for expressing simple constraints on type parameters (for example, `<T: Comparable>` is equivalent to `<T> where T: Comparable` and so on), you can use it to provide more complex constraints on type parameters and their associated types. For example, you can constrain the associated types of type parameters to conform to protocols. For example, `<S: Sequence> where S.Iterator.Element: Equatable` specifies that `S` conforms to the `Sequence` protocol and that the associated type `S.Iterator.Element` conforms to the `Equatable` protocol. This constraint ensures that each element of the sequence is equatable.

You can also specify the requirement that two types be identical, using the `==` operator. For example, `<S1: Sequence, S2: Sequence> where S1.Iterator.Element == S2.Iterator.Element` expresses the constraints that `S1` and `S2` conform to the `Sequence` protocol and that the elements of both sequences must be of the same type.

Any type argument substituted for a type parameter must meet all the constraints and requirements placed on the type parameter.

A generic `where` clause can appear as part of a declaration that includes type parameters, or as part of a declaration that's nested inside of a declaration that includes type parameters. The generic `where` clause for a nested declaration can still refer to the type parameters of the enclosing declaration; however, the requirements from that `where` clause apply only to the declaration where it's written.

If the enclosing declaration also has a `where` clause, the requirements from both clauses are combined. In the example below, `startsWithZero()` is

available only if `Element` conforms to both `SomeProtocol` and `Numeric`.

```
extension Collection where Element: SomeProtocol {
    func startsWithZero() -> Bool where Element: Numeric {
        return first == .zero
    }
}
```

You can overload a generic function or initializer by providing different constraints, requirements, or both on the type parameters. When you call an overloaded generic function or initializer, the compiler uses these constraints to resolve which overloaded function or initializer to invoke.

For more information about generic `where` clauses and to see an example of one in a generic function declaration, see Generic Where Clauses.

GRAMMAR OF A GENERIC PARAMETER CLAUSE

generic-parameter-clause → **<** *generic-parameter-list* **>**
generic-parameter-list → *generic-parameter* | *generic-parameter* **,** *generic-parameter-list*
generic-parameter → *type-name*
generic-parameter → *type-name* **:** *type-identifier*
generic-parameter → *type-name* **:** *protocol-composition-type*
generic-where-clause → **where** *requirement-list*
requirement-list → *requirement* | *requirement* **,** *requirement-list*
requirement → *conformance-requirement* | *same-type-requirement*
conformance-requirement → *type-identifier* **:** *type-identifier*
conformance-requirement → *type-identifier* **:** *protocol-composition-type*
same-type-requirement → *type-identifier* **==** *type*

Generic Argument Clause

A *generic argument clause* specifies the type arguments of a generic type. A generic argument clause is enclosed in angle brackets (<>) and has the following form:

```
<generic argument list>
```

The *generic argument list* is a comma-separated list of type arguments. A *type argument* is the name of an actual concrete type that replaces a corresponding type parameter in the generic parameter clause of a generic type. The result is a specialized version of that generic type. The example below shows a simplified version of the Swift standard library's generic dictionary type.

```
struct Dictionary<Key: Hashable, Value>: Collection,
     ExpressibleByDictionaryLiteral {
    /* ... */
}
```

The specialized version of the generic `Dictionary` type, `Dictionary<String, Int>` is formed by replacing the generic parameters `Key: Hashable` and `Value` with the concrete type arguments `String` and `Int`. Each type argument must satisfy all the constraints of the generic parameter it replaces, including any additional requirements specified in a generic `where` clause. In the example above, the `Key` type parameter is constrained to conform to the `Hashable` protocol and therefore `String` must also conform to the `Hashable` protocol.

You can also replace a type parameter with a type argument that's itself a specialized version of a generic type (provided it satisfies the appropriate constraints and requirements). For example, you can replace the type parameter `Element` in `Array<Element>` with a specialized version of an array, `Array<Int>`, to form an array whose elements are themselves arrays of integers.

```
let arrayOfArrays: Array<Array<Int>> = [[1, 2, 3],
      [4, 5, 6], [7, 8, 9]]
```

As mentioned in Generic Parameter Clause, you don't use a generic argument clause to specify the type arguments of a generic function or initializer.

GRAMMAR OF A GENERIC ARGUMENT CLAUSE

generic-argument-clause → **<** *generic-argument-list* **>**
generic-argument-list → *generic-argument* | *generic-argument* **,** *generic-argument-list*
generic-argument → *type*

Summary of the Grammar

Lexical Structure

GRAMMAR OF WHITESPACE

whitespace → *whitespace-item* *whitespace* *opt*
whitespace-item → *line-break*
whitespace-item → *inline-space*
whitespace-item → *comment*
whitespace-item → *multiline-comment*
whitespace-item → U+0000, U+000B, or U+000C
line-break → U+000A
line-break → U+000D
line-break → U+000D followed by U+000A
inline-spaces → *inline-space* *inline-spaces* *opt*
inline-space → U+0009 or U+0020
comment → `//` *comment-text* *line-break*
multiline-comment → `/*` *multiline-comment-text* `*/`
comment-text → *comment-text-item* *comment-text* *opt*
comment-text-item → Any Unicode scalar value except U+000A or U+000D
multiline-comment-text → *multiline-comment-text-item* *multiline-comment-text* *opt*
multiline-comment-text-item → *multiline-comment*
multiline-comment-text-item → *comment-text-item*
multiline-comment-text-item → Any Unicode scalar value except `/*` or `*/`

GRAMMAR OF AN IDENTIFIER

identifier → *identifier-head* *identifier-characters* *opt*
identifier → ` *identifier-head* *identifier-characters* *opt* `
identifier → *implicit-parameter-name*
identifier → *property-wrapper-projection*
identifier-list → *identifier* | *identifier* **,** *identifier-list*
identifier-head → Upper- or lowercase letter A through Z
identifier-head → _
identifier-head → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2–U+00B5, or U+00B7–U+00BA
identifier-head → U+00BC–U+00BE, U+00C0–U+00D6, U+00D8–U+00F6, or U+00F8–U+00FF
identifier-head → U+0100–U+02FF, U+0370–U+167F, U+1681–U+180D, or U+180F–U+1DBF
identifier-head → U+1E00–U+1FFF
identifier-head → U+200B–U+200D, U+202A–U+202E, U+203F–U+2040, U+2054, or U+2060–U+206F
identifier-head → U+2070–U+20CF, U+2100–U+218F, U+2460–U+24FF, or U+2776–U+2793
identifier-head → U+2C00–U+2DFF or U+2E80–U+2FFF
identifier-head → U+3004–U+3007, U+3021–U+302F, U+3031–U+303F, or U+3040–U+D7FF
identifier-head → U+F900–U+FD3D, U+FD40–U+FDCF, U+FDF0–U+FE1F, or U+FE30–U+FE44
identifier-head → U+FE47–U+FFFD
identifier-head → U+10000–U+1FFFD, U+20000–U+2FFFD, U+30000–U+3FFFD, or U+40000–U+4FFFD
identifier-head → U+50000–U+5FFFD, U+60000–U+6FFFD, U+70000–U+7FFFD, or U+80000–U+8FFFD
identifier-head → U+90000–U+9FFFD, U+A0000–U+AFFFD, U+B0000–U+BFFFD, or U+C0000–U+CFFFD
identifier-head → U+D0000–U+DFFFD or U+E0000–U+EFFFD
identifier-character → Digit 0 through 9
identifier-character → U+0300–U+036F, U+1DC0–U+1DFF, U+20D0–U+20FF, or U+FE20–U+FE2F
identifier-character → *identifier-head*
identifier-characters → *identifier-character* *identifier-characters* *opt*
implicit-parameter-name → **$** *decimal-digits*
property-wrapper-projection → **$** *identifier-characters*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A LITERAL

literal → *numeric-literal* | *string-literal* | *boolean-literal* | *nil-literal*
numeric-literal → **-**$_{opt}$ *integer-literal* | **-**$_{opt}$ *floating-point-literal*
boolean-literal → **true** | **false**
nil-literal → **nil**

GRAMMAR OF AN INTEGER LITERAL

integer-literal → *binary-literal*
integer-literal → *octal-literal*
integer-literal → *decimal-literal*
integer-literal → *hexadecimal-literal*
binary-literal → **0b** *binary-digit* *binary-literal-characters* $_{opt}$
binary-digit → Digit 0 or 1
binary-literal-character → *binary-digit* | **_**
binary-literal-characters → *binary-literal-character* *binary-literal-characters* $_{opt}$
octal-literal → **0o** *octal-digit* *octal-literal-characters* $_{opt}$
octal-digit → Digit 0 through 7
octal-literal-character → *octal-digit* | **_**
octal-literal-characters → *octal-literal-character* *octal-literal-characters* $_{opt}$
decimal-literal → *decimal-digit* *decimal-literal-characters* $_{opt}$
decimal-digit → Digit 0 through 9
decimal-digits → *decimal-digit* *decimal-digits* $_{opt}$
decimal-literal-character → *decimal-digit* | **_**
decimal-literal-characters → *decimal-literal-character* *decimal-literal-characters* $_{opt}$
hexadecimal-literal → **0x** *hexadecimal-digit* *hexadecimal-literal-characters* $_{opt}$
hexadecimal-digit → Digit 0 through 9, a through f, or A through F
hexadecimal-literal-character → *hexadecimal-digit* | **_**
hexadecimal-literal-characters → *hexadecimal-literal-character* *hexadecimal-literal-characters* $_{opt}$

GRAMMAR OF A FLOATING-POINT LITERAL

floating-point-literal → *decimal-literal* *decimal-fraction* *opt* *decimal-exponent* *opt*
floating-point-literal → *hexadecimal-literal* *hexadecimal-fraction* *opt* *hexadecimal-exponent*
decimal-fraction → **.** *decimal-literal*
decimal-exponent → *floating-point-e* *sign* *opt* *decimal-literal*
hexadecimal-fraction → **.** *hexadecimal-digit* *hexadecimal-literal-characters* *opt*
hexadecimal-exponent → *floating-point-p* *sign* *opt* *decimal-literal*
floating-point-e → **e** | **E**
floating-point-p → **p** | **P**
sign → **+** | **–**

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A STRING LITERAL

string-literal → *static-string-literal* | *interpolated-string-literal*
string-literal-opening-delimiter → *extended-string-literal-delimiter* *opt* **"**
string-literal-closing-delimiter → **"** *extended-string-literal-delimiter* *opt*
static-string-literal → *string-literal-opening-delimiter* *quoted-text* *opt* *string-literal-closing-delimiter*
static-string-literal → *multiline-string-literal-opening-delimiter* *multiline-quoted-text* *opt* *multiline-string-literal-closing-delimiter*
multiline-string-literal-opening-delimiter → *extended-string-literal-delimiter* *opt* **"""**
multiline-string-literal-closing-delimiter → **"""** *extended-string-literal-delimiter* *opt*
extended-string-literal-delimiter → **#** *extended-string-literal-delimiter* *opt*
quoted-text → *quoted-text-item* *quoted-text* *opt*
quoted-text-item → *escaped-character*
quoted-text-item → Any Unicode scalar value except **"**, ****, U+000A, or U+000D
multiline-quoted-text → *multiline-quoted-text-item* *multiline-quoted-text* *opt*
multiline-quoted-text-item → *escaped-character*
multiline-quoted-text-item → Any Unicode scalar value except ****
multiline-quoted-text-item → *escaped-newline*
interpolated-string-literal → *string-literal-opening-delimiter* *interpolated-text* *opt* *string-literal-closing-delimiter*
interpolated-string-literal → *multiline-string-literal-opening-delimiter* *multiline-interpolated-text* *opt* *multiline-string-literal-closing-delimiter*
interpolated-text → *interpolated-text-item* *interpolated-text* *opt*
interpolated-text-item → **\(** *expression* **)** | *quoted-text-item*
multiline-interpolated-text → *multiline-interpolated-text-item* *multiline-interpolated-text* *opt*
multiline-interpolated-text-item → **\(** *expression* **)** | *multiline-quoted-text-item*
escape-sequence → **** *extended-string-literal-delimiter*
escaped-character → *escape-sequence* **0** | *escape-sequence* **** | *escape-sequence* **t** | *escape-sequence* **n** | *escape-sequence* **r** | *escape-sequence* **"** | *escape-sequence* **'**
escaped-character → *escape-sequence* **u** **{** *unicode-scalar-digits* **}**
unicode-scalar-digits → Between one and eight hexadecimal digits
escaped-newline → *escape-sequence* *inline-spaces* *opt* *line-break*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF OPERATORS

operator → *operator-head* *operator-characters* *opt*
operator → *dot-operator-head* *dot-operator-characters*
operator-head → **/** | **=** | **-** | **+** | **!** | ***** | **%** | **<** | **>** | **&** | **|** | **^** | **~** | **?**
operator-head → U+00A1–U+00A7
operator-head → U+00A9 or U+00AB
operator-head → U+00AC or U+00AE
operator-head → U+00B0–U+00B1
operator-head → U+00B6, U+00BB, U+00BF, U+00D7, or U+00F7
operator-head → U+2016–U+2017
operator-head → U+2020–U+2027
operator-head → U+2030–U+203E
operator-head → U+2041–U+2053
operator-head → U+2055–U+205E
operator-head → U+2190–U+23FF
operator-head → U+2500–U+2775
operator-head → U+2794–U+2BFF
operator-head → U+2E00–U+2E7F
operator-head → U+3001–U+3003
operator-head → U+3008–U+3020
operator-head → U+3030
operator-character → *operator-head*
operator-character → U+0300–U+036F
operator-character → U+1DC0–U+1DFF
operator-character → U+20D0–U+20FF
operator-character → U+FE00–U+FE0F
operator-character → U+FE20–U+FE2F
operator-character → U+E0100–U+E01EF
operator-characters → *operator-character* *operator-characters* *opt*
dot-operator-head → **.**
dot-operator-character → **.** | *operator-character*
dot-operator-characters → *dot-operator-character* *dot-operator-characters* *opt*
binary-operator → *operator*
prefix-operator → *operator*
postfix-operator → *operator*

Types

GRAMMAR OF A TYPE

type → *function-type*
type → *array-type*
type → *dictionary-type*
type → *type-identifier*
type → *tuple-type*
type → *optional-type*
type → *implicitly-unwrapped-optional-type*
type → *protocol-composition-type*
type → *opaque-type*
type → *metatype-type*
type → *any-type*
type → *self-type*
type → **(** *type* **)**

GRAMMAR OF A TYPE ANNOTATION

type-annotation → **:** *attributes* $_{opt}$ **inout**$_{opt}$ *type*

GRAMMAR OF A TYPE IDENTIFIER

type-identifier → *type-name* *generic-argument-clause* $_{opt}$ | *type-name* *generic-argument-clause* $_{opt}$ **.** *type-identifier*
type-name → *identifier*

GRAMMAR OF A TUPLE TYPE

tuple-type → **( )** | **(** *tuple-type-element* **,** *tuple-type-element-list* **)**
tuple-type-element-list → *tuple-type-element* | *tuple-type-element* **,** *tuple-type-element-list*
tuple-type-element → *element-name* *type-annotation* | *type*
element-name → *identifier*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A FUNCTION TYPE

function-type → *attributes* *opt* *function-type-argument-clause* **throws** *opt* **->** *type*
function-type-argument-clause → **(** **)**
function-type-argument-clause → **(** *function-type-argument-list* **...** *opt* **)**
function-type-argument-list → *function-type-argument* | *function-type-argument* **,** *function-type-argument-list*
function-type-argument → *attributes* *opt* **inout** *opt* *type* | *argument-label* *type-annotation*
argument-label → *identifier*

GRAMMAR OF AN ARRAY TYPE

array-type → **[** *type* **]**

GRAMMAR OF A DICTIONARY TYPE

dictionary-type → **[** *type* **:** *type* **]**

GRAMMAR OF AN OPTIONAL TYPE

optional-type → *type* **?**

GRAMMAR OF AN IMPLICITLY UNWRAPPED OPTIONAL TYPE

implicitly-unwrapped-optional-type → *type* **!**

GRAMMAR OF A PROTOCOL COMPOSITION TYPE

protocol-composition-type → *type-identifier* **&** *protocol-composition-continuation*
protocol-composition-continuation → *type-identifier* | *protocol-composition-type*

GRAMMAR OF AN OPAQUE TYPE

opaque-type → **some** *type*

GRAMMAR OF A METATYPE TYPE

metatype-type → *type* **.** **Type** | *type* **.** **Protocol**

GRAMMAR OF AN ANY TYPE

any-type → **Any**

GRAMMAR OF A SELF TYPE

self-type → **Self**

GRAMMAR OF A TYPE INHERITANCE CLAUSE

type-inheritance-clause → **:** *type-inheritance-list*
type-inheritance-list → *type-identifier* | *type-identifier* **,** *type-inheritance-list*

Expressions

GRAMMAR OF AN EXPRESSION

expression → *try-operator* *opt* *await-operator* *opt* *prefix-expression* *binary-expressions* *opt*
expression-list → *expression* | *expression* **,** *expression-list*

GRAMMAR OF A PREFIX EXPRESSION

prefix-expression → *prefix-operator* *opt* *postfix-expression*
prefix-expression → *in-out-expression*

GRAMMAR OF AN IN-OUT EXPRESSION

in-out-expression → **&** *identifier*

GRAMMAR OF A TRY EXPRESSION

try-operator → **try** | **try ?** | **try !**

GRAMMAR OF AN AWAIT EXPRESSION

await-operator → **await**

GRAMMAR OF A BINARY EXPRESSION

binary-expression → *binary-operator* *prefix-expression*
binary-expression → *assignment-operator* *try-operator* *opt* *prefix-expression*
binary-expression → *conditional-operator* *try-operator* *opt* *prefix-expression*
binary-expression → *type-casting-operator*
binary-expressions → *binary-expression* *binary-expressions* *opt*

GRAMMAR OF AN ASSIGNMENT OPERATOR

assignment-operator → **=**

GRAMMAR OF A CONDITIONAL OPERATOR

conditional-operator → **?** *expression* **:**

GRAMMAR OF A TYPE-CASTING OPERATOR

type-casting-operator → **is** *type*
type-casting-operator → **as** *type*
type-casting-operator → **as ?** *type*
type-casting-operator → **as !** *type*

GRAMMAR OF A PRIMARY EXPRESSION

primary-expression → *identifier* *generic-argument-clause* *opt*
primary-expression → *literal-expression*
primary-expression → *self-expression*
primary-expression → *superclass-expression*
primary-expression → *closure-expression*
primary-expression → *parenthesized-expression*
primary-expression → *tuple-expression*
primary-expression → *implicit-member-expression*
primary-expression → *wildcard-expression*
primary-expression → *key-path-expression*
primary-expression → *selector-expression*
primary-expression → *key-path-string-expression*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A LITERAL EXPRESSION

literal-expression → *literal*
literal-expression → *array-literal* | *dictionary-literal* | *playground-literal*
literal-expression → **#file** | **#fileID** | **#filePath**
literal-expression → **#line** | **#column** | **#function** | **#dsohandle**
array-literal → **[** *array-literal-items* $_{opt}$ **]**
array-literal-items → *array-literal-item* **,** $_{opt}$ | *array-literal-item* **,** *array-literal-items*
array-literal-item → *expression*
dictionary-literal → **[** *dictionary-literal-items* **]** | **[:]**
dictionary-literal-items → *dictionary-literal-item* **,** $_{opt}$ | *dictionary-literal-item* **,** *dictionary-literal-items*
dictionary-literal-item → *expression* **:** *expression*
playground-literal → **#colorLiteral (red :** *expression* **, green :** *expression* **, blue :** *expression* **, alpha :** *expression* **)**
playground-literal → **#fileLiteral (resourceName :** *expression* **)**
playground-literal → **#imageLiteral (resourceName :** *expression* **)**

GRAMMAR OF A SELF EXPRESSION

self-expression → **self** | *self-method-expression* | *self-subscript-expression* | *self-initializer-expression*
self-method-expression → **self .** *identifier*
self-subscript-expression → **self [** *function-call-argument-list* **]**
self-initializer-expression → **self . init**

GRAMMAR OF A SUPERCLASS EXPRESSION

superclass-expression → *superclass-method-expression* | *superclass-subscript-expression* | *superclass-initializer-expression*
superclass-method-expression → **super .** *identifier*
superclass-subscript-expression → **super [** *function-call-argument-list* **]**
superclass-initializer-expression → **super . init**

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A CLOSURE EXPRESSION

closure-expression → **{** *closure-signature* opt *statements* opt **}**
closure-signature → *capture-list* opt *closure-parameter-clause* **throws** opt *function-result* opt **in**
closure-signature → *capture-list* **in**
closure-parameter-clause → **( )** | **(** *closure-parameter-list* **)** | *identifier-list*
closure-parameter-list → *closure-parameter* | *closure-parameter* **,** *closure-parameter-list*
closure-parameter → *closure-parameter-name* *type-annotation* opt
closure-parameter → *closure-parameter-name* *type-annotation* **...**
closure-parameter-name → *identifier*
capture-list → **[** *capture-list-items* **]**
capture-list-items → *capture-list-item* | *capture-list-item* **,** *capture-list-items*
capture-list-item → *capture-specifier* opt *identifier*
capture-list-item → *capture-specifier* opt *identifier* **=** *expression*
capture-list-item → *capture-specifier* opt *self-expression*
capture-specifier → **weak** | **unowned** | **unowned(safe)** | **unowned(unsafe)**

GRAMMAR OF A IMPLICIT MEMBER EXPRESSION

implicit-member-expression → **.** *identifier*
implicit-member-expression → **.** *identifier* **.** *postfix-expression*

GRAMMAR OF A PARENTHESIZED EXPRESSION

parenthesized-expression → **(** *expression* **)**

GRAMMAR OF A TUPLE EXPRESSION

tuple-expression → **( )** | **(** *tuple-element* **,** *tuple-element-list* **)**
tuple-element-list → *tuple-element* | *tuple-element* **,** *tuple-element-list*
tuple-element → *expression* | *identifier* **:** *expression*

GRAMMAR OF A WILDCARD EXPRESSION

wildcard-expression → **_**

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A KEY-PATH EXPRESSION

key-path-expression → \ *type* $_{opt}$. *key-path-components*
key-path-components → *key-path-component* | *key-path-component* . *key-path-components*
key-path-component → *identifier* *key-path-postfixes* $_{opt}$ | *key-path-postfixes*
key-path-postfixes → *key-path-postfix* *key-path-postfixes* $_{opt}$
key-path-postfix → **?** | **!** | **self** | **[** *function-call-argument-list* **]**

GRAMMAR OF A SELECTOR EXPRESSION

selector-expression → **#selector** **(** *expression* **)**
selector-expression → **#selector** **(** **getter:** *expression* **)**
selector-expression → **#selector** **(** **setter:** *expression* **)**

GRAMMAR OF A KEY-PATH STRING EXPRESSION

key-path-string-expression → **#keyPath** **(** *expression* **)**

GRAMMAR OF A POSTFIX EXPRESSION

postfix-expression → *primary-expression*
postfix-expression → *postfix-expression* *postfix-operator*
postfix-expression → *function-call-expression*
postfix-expression → *initializer-expression*
postfix-expression → *explicit-member-expression*
postfix-expression → *postfix-self-expression*
postfix-expression → *subscript-expression*
postfix-expression → *forced-value-expression*
postfix-expression → *optional-chaining-expression*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A FUNCTION CALL EXPRESSION

function-call-expression → *postfix-expression* *function-call-argument-clause*
function-call-expression → *postfix-expression* *function-call-argument-clause* *opt* *trailing-closures*
function-call-argument-clause → **(** **)** | **(** *function-call-argument-list* **)**
function-call-argument-list → *function-call-argument* | *function-call-argument* **,** *function-call-argument-list*
function-call-argument → *expression* | *identifier* **:** *expression*
function-call-argument → *operator* | *identifier* **:** *operator*
trailing-closures → *closure-expression* *labeled-trailing-closures* *opt*
labeled-trailing-closures → *labeled-trailing-closure* *labeled-trailing-closures* *opt*
labeled-trailing-closure → *identifier* **:** *closure-expression*

GRAMMAR OF AN INITIALIZER EXPRESSION

initializer-expression → *postfix-expression* **. init**
initializer-expression → *postfix-expression* **. init** **(** *argument-names* **)**

GRAMMAR OF AN EXPLICIT MEMBER EXPRESSION

explicit-member-expression → *postfix-expression* **.** *decimal-digits*
explicit-member-expression → *postfix-expression* **.** *identifier* *generic-argument-clause* *opt*
explicit-member-expression → *postfix-expression* **.** *identifier* **(** *argument-names* **)**
argument-names → *argument-name* *argument-names* *opt*
argument-name → *identifier* **:**

GRAMMAR OF A POSTFIX SELF EXPRESSION

postfix-self-expression → *postfix-expression* **. self**

GRAMMAR OF A SUBSCRIPT EXPRESSION

subscript-expression → *postfix-expression* **[** *function-call-argument-list* **]**

GRAMMAR OF A FORCED-VALUE EXPRESSION

forced-value-expression → *postfix-expression* **!**

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF AN OPTIONAL-CHAINING EXPRESSION

optional-chaining-expression → *postfix-expression* **?**

Statements

GRAMMAR OF A STATEMENT

statement → *expression* **;** *opt*
statement → *declaration* **;** *opt*
statement → *loop-statement* **;** *opt*
statement → *branch-statement* **;** *opt*
statement → *labeled-statement* **;** *opt*
statement → *control-transfer-statement* **;** *opt*
statement → *defer-statement* **;** *opt*
statement → *do-statement* **;** *opt*
statement → *compiler-control-statement*
statements → *statement* *statements* *opt*

GRAMMAR OF A LOOP STATEMENT

loop-statement → *for-in-statement*
loop-statement → *while-statement*
loop-statement → *repeat-while-statement*

GRAMMAR OF A FOR-IN STATEMENT

for-in-statement → **for** **case** *opt* *pattern* **in** *expression* *where-clause* *opt* *code-block*

GRAMMAR OF A WHILE STATEMENT

while-statement → **while** *condition-list* *code-block*
condition-list → *condition* | *condition* **,** *condition-list*
condition → *expression* | *availability-condition* | *case-condition* | *optional-binding-condition*
case-condition → **case** *pattern* *initializer*
optional-binding-condition → **let** *pattern* *initializer* | **var** *pattern* *initializer*

GRAMMAR OF A REPEAT-WHILE STATEMENT

repeat-while-statement → **repeat** *code-block* **while** *expression*

GRAMMAR OF A BRANCH STATEMENT

branch-statement → *if-statement*
branch-statement → *guard-statement*
branch-statement → *switch-statement*

GRAMMAR OF AN IF STATEMENT

if-statement → **if** *condition-list* *code-block* *else-clause* *opt*
else-clause → **else** *code-block* | **else** *if-statement*

GRAMMAR OF A GUARD STATEMENT

guard-statement → **guard** *condition-list* **else** *code-block*

GRAMMAR OF A SWITCH STATEMENT

switch-statement → **switch** *expression* **{** *switch-cases* *opt* **}**
switch-cases → *switch-case* *switch-cases* *opt*
switch-case → *case-label* *statements*
switch-case → *default-label* *statements*
switch-case → *conditional-switch-case*
case-label → *attributes* *opt* **case** *case-item-list* **:**
case-item-list → *pattern* *where-clause* *opt* | *pattern* *where-clause* *opt* **,** *case-item-list*
default-label → *attributes* *opt* **default** **:**
where-clause → **where** *where-expression*
where-expression → *expression*
conditional-switch-case → *switch-if-directive-clause* *switch-elseif-directive-clauses* *opt* *switch-else-directive-clause* *opt* *endif-directive*
switch-if-directive-clause → *if-directive* *compilation-condition* *switch-cases* *opt*
switch-elseif-directive-clauses → *elseif-directive-clause* *switch-elseif-directive-clauses* *opt*
switch-elseif-directive-clause → *elseif-directive* *compilation-condition* *switch-cases* *opt*
switch-else-directive-clause → *else-directive* *switch-cases* *opt*

GRAMMAR OF A LABELED STATEMENT

labeled-statement → *statement-label* *loop-statement*
labeled-statement → *statement-label* *if-statement*
labeled-statement → *statement-label* *switch-statement*
labeled-statement → *statement-label* *do-statement*
statement-label → *label-name* **:**
label-name → *identifier*

GRAMMAR OF A CONTROL TRANSFER STATEMENT

control-transfer-statement → *break-statement*
control-transfer-statement → *continue-statement*
control-transfer-statement → *fallthrough-statement*
control-transfer-statement → *return-statement*
control-transfer-statement → *throw-statement*

GRAMMAR OF A BREAK STATEMENT

break-statement → **break** *label-name* *opt*

GRAMMAR OF A CONTINUE STATEMENT

continue-statement → **continue** *label-name* *opt*

GRAMMAR OF A FALLTHROUGH STATEMENT

fallthrough-statement → **fallthrough**

GRAMMAR OF A RETURN STATEMENT

return-statement → **return** *expression* *opt*

GRAMMAR OF A THROW STATEMENT

throw-statement → **throw** *expression*

GRAMMAR OF A DEFER STATEMENT

defer-statement → **defer** *code-block*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A DO STATEMENT

do-statement → **do** *code-block* *catch-clauses* *opt*
catch-clauses → *catch-clause* *catch-clauses* *opt*
catch-clause → **catch** *catch-pattern-list* *opt* *code-block*
catch-pattern-list → *catch-pattern* | *catch-pattern* **,** *catch-pattern-list*
catch-pattern → *pattern* *where-clause* *opt*

GRAMMAR OF A COMPILER CONTROL STATEMENT

compiler-control-statement → *conditional-compilation-block*
compiler-control-statement → *line-control-statement*
compiler-control-statement → *diagnostic-statement*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A CONDITIONAL COMPILATION BLOCK

conditional-compilation-block → *if-directive-clause* *elseif-directive-clauses* *opt* *else-directive-clause* *opt* *endif-directive*
if-directive-clause → *if-directive* *compilation-condition* *statements* *opt*
elseif-directive-clauses → *elseif-directive-clause* *elseif-directive-clauses* *opt*
elseif-directive-clause → *elseif-directive* *compilation-condition* *statements* *opt*
else-directive-clause → *else-directive* *statements* *opt*
if-directive → **#if**
elseif-directive → **#elseif**
else-directive → **#else**
endif-directive → **#endif**
compilation-condition → *platform-condition*
compilation-condition → *identifier*
compilation-condition → *boolean-literal*
compilation-condition → **(** *compilation-condition* **)**
compilation-condition → **!** *compilation-condition*
compilation-condition → *compilation-condition* **&&** *compilation-condition*
compilation-condition → *compilation-condition* **||** *compilation-condition*
platform-condition → **os** **(** *operating-system* **)**
platform-condition → **arch** **(** *architecture* **)**
platform-condition → **swift** **(** **>=** *swift-version* **)** | **swift** **(** **<** *swift-version* **)**
platform-condition → **compiler** **(** **>=** *swift-version* **)** | **compiler** **(** **<** *swift-version* **)**
platform-condition → **canImport** **(** *module-name* **)**
platform-condition → **targetEnvironment** **(** *environment* **)**
operating-system → **macOS** | **iOS** | **watchOS** | **tvOS** | **Linux** | **Windows**
architecture → **i386** | **x86_64** | **arm** | **arm64**
swift-version → *decimal-digits* *swift-version-continuation* *opt*
swift-version-continuation → **.** *decimal-digits* *swift-version-continuation* *opt*
module-name → *identifier*
environment → **simulator** | **macCatalyst**

GRAMMAR OF A LINE CONTROL STATEMENT

line-control-statement → **#sourceLocation** **(** **file:** *file-path* **,** **line:** *line-number* **)**
line-control-statement → **#sourceLocation** **(** **)**
line-number → A decimal integer greater than zero
file-path → *static-string-literal*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A COMPILE-TIME DIAGNOSTIC STATEMENT

diagnostic-statement → **#error** **(** *diagnostic-message* **)**
diagnostic-statement → **#warning** **(** *diagnostic-message* **)**
diagnostic-message → *static-string-literal*

GRAMMAR OF AN AVAILABILITY CONDITION

availability-condition → **#available** **(** *availability-arguments* **)**
availability-arguments → *availability-argument* | *availability-argument* **,** *availability-arguments*
availability-argument → *platform-name* *platform-version*
availability-argument → *****
platform-name → **iOS** | **iOSApplicationExtension**
platform-name → **macOS** | **macOSApplicationExtension**
platform-name → **macCatalyst** | **macCatalystApplicationExtension**
platform-name → **watchOS**
platform-name → **tvOS**
platform-version → *decimal-digits*
platform-version → *decimal-digits* **.** *decimal-digits*
platform-version → *decimal-digits* **.** *decimal-digits* **.** *decimal-digits*

Declarations

GRAMMAR OF A DECLARATION

declaration → *import-declaration*
declaration → *constant-declaration*
declaration → *variable-declaration*
declaration → *typealias-declaration*
declaration → *function-declaration*
declaration → *enum-declaration*
declaration → *struct-declaration*
declaration → *class-declaration*
declaration → *actor-declaration*
declaration → *protocol-declaration*
declaration → *initializer-declaration*
declaration → *deinitializer-declaration*
declaration → *extension-declaration*
declaration → *subscript-declaration*
declaration → *operator-declaration*
declaration → *precedence-group-declaration*
declarations → *declaration* *declarations* *opt*

GRAMMAR OF A TOP-LEVEL DECLARATION

top-level-declaration → *statements* *opt*

GRAMMAR OF A CODE BLOCK

code-block → **{** *statements* *opt* **}**

GRAMMAR OF AN IMPORT DECLARATION

import-declaration → *attributes* *opt* **import** *import-kind* *opt* *import-path*
import-kind → **typealias** | **struct** | **class** | **enum** | **protocol** | **let** | **var** | **func**
import-path → *import-path-identifier* | *import-path-identifier* **.** *import-path*
import-path-identifier → *identifier* | *operator*

GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → *attributes* *opt* *declaration-modifiers* *opt* **let** *pattern-initializer-list*
pattern-initializer-list → *pattern-initializer* | *pattern-initializer* **,** *pattern-initializer-list*
pattern-initializer → *pattern* *initializer* *opt*
initializer → **=** *expression*

GRAMMAR OF A VARIABLE DECLARATION

variable-declaration → *variable-declaration-head* *pattern-initializer-list*
variable-declaration → *variable-declaration-head* *variable-name* *type-annotation* *code-block*
variable-declaration → *variable-declaration-head* *variable-name* *type-annotation* *getter-setter-block*
variable-declaration → *variable-declaration-head* *variable-name* *type-annotation* *getter-setter-keyword-block*
variable-declaration → *variable-declaration-head* *variable-name* *initializer* *willSet-didSet-block*
variable-declaration → *variable-declaration-head* *variable-name* *type-annotation* *initializer* *opt* *willSet-didSet-block*
variable-declaration-head → *attributes* *opt* *declaration-modifiers* *opt* **var**
variable-name → *identifier*
getter-setter-block → *code-block*
getter-setter-block → **{** *getter-clause* *setter-clause* *opt* **}**
getter-setter-block → **{** *setter-clause* *getter-clause* **}**
getter-clause → *attributes* *opt* *mutation-modifier* *opt* **get** *code-block*
setter-clause → *attributes* *opt* *mutation-modifier* *opt* **set** *setter-name* *opt* *code-block*
setter-name → **(** *identifier* **)**
getter-setter-keyword-block → **{** *getter-keyword-clause* *setter-keyword-clause* *opt* **}**
getter-setter-keyword-block → **{** *setter-keyword-clause* *getter-keyword-clause* **}**
getter-keyword-clause → *attributes* *opt* *mutation-modifier* *opt* **get**
setter-keyword-clause → *attributes* *opt* *mutation-modifier* *opt* **set**
willSet-didSet-block → **{** *willSet-clause* *didSet-clause* *opt* **}**
willSet-didSet-block → **{** *didSet-clause* *willSet-clause* *opt* **}**
willSet-clause → *attributes* *opt* **willSet** *setter-name* *opt* *code-block*
didSet-clause → *attributes* *opt* **didSet** *setter-name* *opt* *code-block*

GRAMMAR OF A TYPE ALIAS DECLARATION

typealias-declaration → *attributes* *opt* *access-level-modifier* *opt* **typealias** *typealias-name* *generic-parameter-clause* *opt* *typealias-assignment*
typealias-name → *identifier*
typealias-assignment → **=** *type*

GRAMMAR OF A FUNCTION DECLARATION

function-declaration → *function-head* *function-name* *generic-parameter-clause* *opt* *function-signature* *generic-where-clause* *opt* *function-body* *opt*
function-head → *attributes* *opt* *declaration-modifiers* *opt* **func**
function-name → *identifier* | *operator*
function-signature → *parameter-clause* **async** *opt* **throws** *opt* *function-result* *opt*
function-signature → *parameter-clause* **async** *opt* **rethrows** *function-result* *opt*
function-result → **->** *attributes* *opt* *type*
function-body → *code-block*
parameter-clause → **( )** | **(** *parameter-list* **)**
parameter-list → *parameter* | *parameter* **,** *parameter-list*
parameter → *external-parameter-name* *opt* *local-parameter-name* *type-annotation* *default-argument-clause* *opt*
parameter → *external-parameter-name* *opt* *local-parameter-name* *type-annotation*
parameter → *external-parameter-name* *opt* *local-parameter-name* *type-annotation* **...**
external-parameter-name → *identifier*
local-parameter-name → *identifier*
default-argument-clause → **=** *expression*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF AN ENUMERATION DECLARATION

enum-declaration → *attributes* *opt* *access-level-modifier* *opt* *union-style-enum*
enum-declaration → *attributes* *opt* *access-level-modifier* *opt* *raw-value-style-enum*
union-style-enum → **indirect** *opt* **enum** *enum-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* **{** *union-style-enum-members* *opt* **}**
union-style-enum-members → *union-style-enum-member* *union-style-enum-members* *opt*
union-style-enum-member → *declaration* | *union-style-enum-case-clause* | *compiler-control-statement*
union-style-enum-case-clause → *attributes* *opt* **indirect** *opt* **case** *union-style-enum-case-list*
union-style-enum-case-list → *union-style-enum-case* | *union-style-enum-case* **,** *union-style-enum-case-list*
union-style-enum-case → *enum-case-name* *tuple-type* *opt*
enum-name → *identifier*
enum-case-name → *identifier*
raw-value-style-enum → **enum** *enum-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *generic-where-clause* *opt* **{** *raw-value-style-enum-members* **}**
raw-value-style-enum-members → *raw-value-style-enum-member* *raw-value-style-enum-members* *opt*
raw-value-style-enum-member → *declaration* | *raw-value-style-enum-case-clause* | *compiler-control-statement*
raw-value-style-enum-case-clause → *attributes* *opt* **case** *raw-value-style-enum-case-list*
raw-value-style-enum-case-list → *raw-value-style-enum-case* | *raw-value-style-enum-case* **,** *raw-value-style-enum-case-list*
raw-value-style-enum-case → *enum-case-name* *raw-value-assignment* *opt*
raw-value-assignment → **=** *raw-value-literal*
raw-value-literal → *numeric-literal* | *static-string-literal* | *boolean-literal*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A STRUCTURE DECLARATION

struct-declaration → *attributes* *opt* *access-level-modifier* *opt* **struct** *struct-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *struct-body*
struct-name → *identifier*
struct-body → **{** *struct-members* *opt* **}**
struct-members → *struct-member* *struct-members* *opt*
struct-member → *declaration* | *compiler-control-statement*

GRAMMAR OF A CLASS DECLARATION

class-declaration → *attributes* *opt* *access-level-modifier* *opt* **final** *opt* **class** *class-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *class-body*
class-declaration → *attributes* *opt* **final** *access-level-modifier* *opt* **class** *class-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *class-body*
class-name → *identifier*
class-body → **{** *class-members* *opt* **}**
class-members → *class-member* *class-members* *opt*
class-member → *declaration* | *compiler-control-statement*

GRAMMAR OF AN ACTOR DECLARATION

actor-declaration → *attributes* *opt* *access-level-modifier* *opt* **actor** *actor-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *actor-body*
actor-name → *identifier*
actor-body → **{** *actor-members* *opt* **}**
actor-members → *actor-member* *actor-members* *opt*
actor-member → *declaration* | *compiler-control-statement*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A PROTOCOL DECLARATION

protocol-declaration → *attributes* *opt* *access-level-modifier* *opt* **protocol** *protocol-name* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *protocol-body*
protocol-name → *identifier*
protocol-body → **{** *protocol-members* *opt* **}**
protocol-members → *protocol-member* *protocol-members* *opt*
protocol-member → *protocol-member-declaration* | *compiler-control-statement*
protocol-member-declaration → *protocol-property-declaration*
protocol-member-declaration → *protocol-method-declaration*
protocol-member-declaration → *protocol-initializer-declaration*
protocol-member-declaration → *protocol-subscript-declaration*
protocol-member-declaration → *protocol-associated-type-declaration*
protocol-member-declaration → *typealias-declaration*

GRAMMAR OF A PROTOCOL PROPERTY DECLARATION

protocol-property-declaration → *variable-declaration-head* *variable-name* *type-annotation* *getter-setter-keyword-block*

GRAMMAR OF A PROTOCOL METHOD DECLARATION

protocol-method-declaration → *function-head* *function-name* *generic-parameter-clause* *opt* *function-signature* *generic-where-clause* *opt*

GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION

protocol-initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause* **throws** *opt* *generic-where-clause* *opt*
protocol-initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause* **rethrows** *generic-where-clause* *opt*

GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION

protocol-subscript-declaration → *subscript-head* *subscript-result* *generic-where-clause* *opt* *getter-setter-keyword-block*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION

protocol-associated-type-declaration → *attributes* *opt* *access-level-modifier* *opt* **associatedtype** *typealias-name* *type-inheritance-clause* *opt* *typealias-assignment* *opt* *generic-where-clause* *opt*

GRAMMAR OF AN INITIALIZER DECLARATION

initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause* **throws** *opt* *generic-where-clause* *opt* *initializer-body*
initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause* **rethrows** *generic-where-clause* *opt* *initializer-body*
initializer-head → *attributes* *opt* *declaration-modifiers* *opt* **init**
initializer-head → *attributes* *opt* *declaration-modifiers* *opt* **init ?**
initializer-head → *attributes* *opt* *declaration-modifiers* *opt* **init !**
initializer-body → *code-block*

GRAMMAR OF A DEINITIALIZER DECLARATION

deinitializer-declaration → *attributes* *opt* **deinit** *code-block*

GRAMMAR OF AN EXTENSION DECLARATION

extension-declaration → *attributes* *opt* *access-level-modifier* *opt* **extension** *type-identifier* *type-inheritance-clause* *opt* *generic-where-clause* *opt* *extension-body*
extension-body → **{** *extension-members* *opt* **}**
extension-members → *extension-member* *extension-members* *opt*
extension-member → *declaration* | *compiler-control-statement*

GRAMMAR OF A SUBSCRIPT DECLARATION

subscript-declaration → *subscript-head* *subscript-result* *generic-where-clause* *opt* *code-block*
subscript-declaration → *subscript-head* *subscript-result* *generic-where-clause* *opt* *getter-setter-block*
subscript-declaration → *subscript-head* *subscript-result* *generic-where-clause* *opt* *getter-setter-keyword-block*
subscript-head → *attributes* *opt* *declaration-modifiers* *opt* **subscript** *generic-parameter-clause* *opt* *parameter-clause*
subscript-result → **->** *attributes* *opt* *type*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF AN OPERATOR DECLARATION

operator-declaration → *prefix-operator-declaration* | *postfix-operator-declaration* | *infix-operator-declaration*
prefix-operator-declaration → **prefix operator** *operator*
postfix-operator-declaration → **postfix operator** *operator*
infix-operator-declaration → **infix operator** *operator* *infix-operator-group* *opt*
infix-operator-group → **:** *precedence-group-name*

GRAMMAR OF A PRECEDENCE GROUP DECLARATION

precedence-group-declaration → **precedencegroup** *precedence-group-name* **{** *precedence-group-attributes* *opt* **}**
precedence-group-attributes → *precedence-group-attribute* *precedence-group-attributes* *opt*
precedence-group-attribute → *precedence-group-relation*
precedence-group-attribute → *precedence-group-assignment*
precedence-group-attribute → *precedence-group-associativity*
precedence-group-relation → **higherThan :** *precedence-group-names*
precedence-group-relation → **lowerThan :** *precedence-group-names*
precedence-group-assignment → **assignment :** *boolean-literal*
precedence-group-associativity → **associativity : left**
precedence-group-associativity → **associativity : right**
precedence-group-associativity → **associativity : none**
precedence-group-names → *precedence-group-name* | *precedence-group-name* **,** *precedence-group-names*
precedence-group-name → *identifier*

Converted by Evan at Apps Dissected -www.appsdissected.com

GRAMMAR OF A DECLARATION MODIFIER

declaration-modifier → **class** | **convenience** | **dynamic** | **final** | **infix** | **lazy** | **optional** | **override** | **postfix** | **prefix** | **required** | **static** | **unowned** | **unowned** **(** **safe** **)** | **unowned** **(** **unsafe** **)** | **weak**
declaration-modifier → *access-level-modifier*
declaration-modifier → *mutation-modifier*
declaration-modifier → *actor-isolation-modifier*
declaration-modifiers → *declaration-modifier* *declaration-modifiers* *opt*
access-level-modifier → **private** | **private** **(** **set** **)**
access-level-modifier → **fileprivate** | **fileprivate** **(** **set** **)**
access-level-modifier → **internal** | **internal** **(** **set** **)**
access-level-modifier → **public** | **public** **(** **set** **)**
access-level-modifier → **open** | **open** **(** **set** **)**
mutation-modifier → **mutating** | **nonmutating**
actor-isolation-modifier → **nonisolated**

Attributes

GRAMMAR OF AN ATTRIBUTE

attribute → **@** *attribute-name* *attribute-argument-clause* *opt*
attribute-name → *identifier*
attribute-argument-clause → **(** *balanced-tokens* *opt* **)**
attributes → *attribute* *attributes* *opt*
balanced-tokens → *balanced-token* *balanced-tokens* *opt*
balanced-token → **(** *balanced-tokens* *opt* **)**
balanced-token → **[** *balanced-tokens* *opt* **]**
balanced-token → **{** *balanced-tokens* *opt* **}**
balanced-token → Any identifier, keyword, literal, or operator
balanced-token → Any punctuation except **(**, **)**, **[**, **]**, **{**, or **}**

Patterns

GRAMMAR OF A PATTERN

pattern → *wildcard-pattern* *type-annotation* *opt*
pattern → *identifier-pattern* *type-annotation* *opt*
pattern → *value-binding-pattern*
pattern → *tuple-pattern* *type-annotation* *opt*
pattern → *enum-case-pattern*
pattern → *optional-pattern*
pattern → *type-casting-pattern*
pattern → *expression-pattern*

GRAMMAR OF A WILDCARD PATTERN

wildcard-pattern → **_**

GRAMMAR OF AN IDENTIFIER PATTERN

identifier-pattern → *identifier*

GRAMMAR OF A VALUE-BINDING PATTERN

value-binding-pattern → **var** *pattern* | **let** *pattern*

GRAMMAR OF A TUPLE PATTERN

tuple-pattern → **(** *tuple-pattern-element-list* *opt* **)**
tuple-pattern-element-list → *tuple-pattern-element* | *tuple-pattern-element* **,** *tuple-pattern-element-list*
tuple-pattern-element → *pattern* | *identifier* **:** *pattern*

GRAMMAR OF AN ENUMERATION CASE PATTERN

enum-case-pattern → *type-identifier* *opt* **.** *enum-case-name* *tuple-pattern* *opt*

GRAMMAR OF AN OPTIONAL PATTERN

optional-pattern → *identifier-pattern* **?**

GRAMMAR OF A TYPE CASTING PATTERN

type-casting-pattern → *is-pattern* | *as-pattern*
is-pattern → **is** *type*
as-pattern → *pattern* **as** *type*

GRAMMAR OF AN EXPRESSION PATTERN

expression-pattern → *expression*

Generic Parameters and Arguments

GRAMMAR OF A GENERIC PARAMETER CLAUSE

generic-parameter-clause → **<** *generic-parameter-list* **>**
generic-parameter-list → *generic-parameter* | *generic-parameter* **,** *generic-parameter-list*
generic-parameter → *type-name*
generic-parameter → *type-name* **:** *type-identifier*
generic-parameter → *type-name* **:** *protocol-composition-type*
generic-where-clause → **where** *requirement-list*
requirement-list → *requirement* | *requirement* **,** *requirement-list*
requirement → *conformance-requirement* | *same-type-requirement*
conformance-requirement → *type-identifier* **:** *type-identifier*
conformance-requirement → *type-identifier* **:** *protocol-composition-type*
same-type-requirement → *type-identifier* **==** *type*

GRAMMAR OF A GENERIC ARGUMENT CLAUSE

generic-argument-clause → **<** *generic-argument-list* **>**
generic-argument-list → *generic-argument* | *generic-argument* **,** *generic-argument-list*
generic-argument → *type*

Revision History

Converted by Evan at Apps Dissected -www.appsdissected.com

Document Revision History

2021-09-20

- Updated for Swift 5.5.
- Added information about asynchronous functions, tasks, and actors to the Concurrency chapter, and to the Actor Declaration, Asynchronous Functions and Methods, and Await Operator sections.
- Updated the Identifiers section with information about identifiers that start with an underscore.

2021-04-26

- Updated for Swift 5.4.
- Added the Result Builders and resultBuilder sections with information about result builders.
- Added the Implicit Conversion to a Pointer Type section with information about how in-out parameters can be implicitly converted to unsafe pointers in a function call.
- Updated the Variadic Parameters and Function Declaration sections, now that a function can have multiple variadic parameters.
- Updated the Implicit Member Expression section, now that implicit member expressions can be chained together.

2020-09-16

- Updated for Swift 5.3.
- Added information about multiple trailing closures to the Trailing Closures section, and added information about how trailing closures

are matched to parameters to the Function Call Expression section.

- Added information about synthesized implementations of `Comparable` for enumerations to the Adopting a Protocol Using a Synthesized Implementation section.
- Added the Contextual Where Clauses section now that you can write a generic `where` clause in more places.
- Added the Unowned Optional References section with information about using unowned references with optional values.
- Added information about the `@main` attribute to the main section.
- Added `#filePath` to the Literal Expression section, and updated the discussion of `#file`.
- Updated the Escaping Closures section, now that closures can refer to `self` implicitly in more scenarios.
- Updated the Handling Errors Using Do-Catch and Do Statement sections, now that a `catch` clause can match against multiple errors.
- Added more information about `Any` and moved it into the new Any Type section.
- Updated the Property Observers section, now that lazy properties can have observers.
- Updated the Protocol Declaration section, now that members of an enumeration can satisfy protocol requirements.
- Updated the Stored Variable Observers and Property Observers section to describe when the getter is called before the observer.
- Updated the Memory Safety chapter to mention atomic operations.

2020-03-24

- Updated for Swift 5.2.
- Added information about passing a key path instead of a closure to the Key-Path Expression section.
- Added the Methods with Special Names section with information about syntactic sugar the lets instances of classes, structures, and enumerations be used with function call syntax.
- Updated the Subscript Options section, now that subscripts support parameters with default values.
- Updated the Self Type section, now that the `Self` can be used in more contexts.
- Updated the Implicitly Unwrapped Optionals section to make it clearer that an implicitly unwrapped optional value can be used as either an optional or non-optional value.

2019-09-10

- Updated for Swift 5.1.
- Added information about functions that specify a protocol that their return value conforms to, instead of providing a specific named return type, to the Opaque Types chapter.
- Added information about property wrappers to the Property Wrappers section.
- Added information about enumerations and structures that are frozen for library evolution to the frozen section.
- Added the Functions With an Implicit Return and Shorthand Getter Declaration sections with information about functions that omit `return`.

- Added information about using subscripts on types to the Type Subscripts section.
- Updated the Enumeration Case Pattern section, now that an enumeration case pattern can match an optional value.
- Updated the Memberwise Initializers for Structure Types section, now that memberwise initializers support omitting parameters for properties that have a default value.
- Added information about dynamic members that are looked up by key path at runtime to the dynamicMemberLookup section.
- Added `macCatalyst` to the list of target environments in Conditional Compilation Block.
- Updated the Self Type section, now that `Self` can be used to refer to the type introduced by the current class, structure, or enumeration declaration.

2019-03-25

- Updated for Swift 5.0.
- Added the Extended String Delimiters section and updated the String Literals section with information about extended string delimiters.
- Added the dynamicCallable section with information about dynamically calling instances as functions using the `dynamicCallable` attribute.
- Added the unknown and Switching Over Future Enumeration Cases sections with information about handling future enumeration cases in switch statements using the `unknown` switch case attribute.
- Added information about the identity key path (`\.self`) to the Key-Path Expression section.

- Added information about using the less than (<) operator in platform conditions to the Conditional Compilation Block section.

2018-09-17

- Updated for Swift 4.2.
- Added information about accessing all of an enumeration's cases to the Iterating over Enumeration Cases section.
- Added information about `#error` and `#warning` to the Compile-Time Diagnostic Statement section.
- Added information about inlining to the Declaration Attributes section under the `inlinable` and `usableFromInline` attributes.
- Added information about members that are looked up by name at runtime to the Declaration Attributes section under the `dynamicMemberLookup` attribute.
- Added information about the `requires_stored_property_inits` and `warn_unqualified_access` attributes to the Declaration Attributes section.
- Added information about how to conditionally compile code depending on the Swift compiler version being used to the Conditional Compilation Block section.
- Added information about `#dsohandle` to the Literal Expression section.

2018-03-29

- Updated for Swift 4.1.
- Added information about synthesized implementations of equivalence operators to the Equivalence Operators section.

- Added information about conditional protocol conformance to the Extension Declaration section of the Declarations chapter, and to the Conditionally Conforming to a Protocol section of the Protocols chapter.

- Added information about recursive protocol constraints to the Using a Protocol in Its Associated Type's Constraints section.

- Added information about the `canImport()` and `targetEnvironment()` platform conditions to Conditional Compilation Block.

2017-12-04

- Updated for Swift 4.0.3.

- Updated the Key-Path Expression section, now that key paths support subscript components.

2017-09-19

- Updated for Swift 4.0.

- Added information about exclusive access to memory to the Memory Safety chapter.

- Added the Associated Types with a Generic Where Clause section, now that you can use generic `where` clauses to constrain associated types.

- Added information about multiline string literals to the String Literals section of the Strings and Characters chapter, and to the String Literals section of the Lexical Structure chapter.

- Updated the discussion of the `objc` attribute in Declaration Attributes, now that this attribute is inferred in fewer places.

- Added the Generic Subscripts section, now that subscripts can be generic.
- Updated the discussion in the Protocol Composition section of the Protocols chapter, and in the Protocol Composition Type section of the Types chapter, now that protocol composition types can contain a superclass requirement.
- Updated the discussion of protocol extensions in Extension Declaration now that `final` isn't allowed in them.
- Added information about preconditions and fatal errors to the Assertions and Preconditions section.

2017-03-27

- Updated for Swift 3.1.
- Added the Extensions with a Generic Where Clause section with information about extensions that include requirements.
- Added examples of iterating over a range to the For-In Loops section.
- Added an example of failable numeric conversions to the Failable Initializers section.
- Added information to the Declaration Attributes section about using the `available` attribute with a Swift language version.
- Updated the discussion in the Function Type section to note that argument labels aren't allowed when writing a function type.
- Updated the discussion of Swift language version numbers in the Conditional Compilation Block section, now that an optional patch number is allowed.
- Updated the discussion in the Function Type section, now that Swift distinguishes between functions that take multiple parameters and

functions that take a single parameter of a tuple type.

- Removed the Dynamic Type Expression section from the Expressions chapter, now that `type(of:)` is a Swift standard library function.

2016-10-27

- Updated for Swift 3.0.1.
- Updated the discussion of weak and unowned references in the Automatic Reference Counting chapter.
- Added information about the `unowned`, `unowned(safe)`, and `unowned(unsafe)` declaration modifiers in the Declaration Modifiers section.
- Added a note to the Type Casting for Any and AnyObject section about using an optional value when a value of type `Any` is expected.
- Updated the Expressions chapter to separate the discussion of parenthesized expressions and tuple expressions.

2016-09-13

- Updated for Swift 3.0.
- Updated the discussion of functions in the Functions chapter and the Function Declaration section to note that all parameters get an argument label by default.
- Updated the discussion of operators in the Advanced Operators chapter, now that you implement them as type methods instead of as global functions.
- Added information about the `open` and `fileprivate` access-level modifiers to the Access Control chapter.

- Updated the discussion of `inout` in the Function Declaration section to note that it appears in front of a parameter's type instead of in front of a parameter's name.

- Updated the discussion of the `@noescape` and `@autoclosure` attributes in the Escaping Closures and Autoclosures sections and the Attributes chapter now that they're type attributes, rather than declaration attributes.

- Added information about operator precedence groups to the Precedence for Custom Infix Operators section of the Advanced Operators chapter, and to the Precedence Group Declaration section of the Declarations chapter.

- Updated discussion throughout to use macOS instead of OS X, `Error` instead of `ErrorProtocol`, and protocol names such as `ExpressibleByStringLiteral` instead of `StringLiteralConvertible`.

- Updated the discussion in the Generic Where Clauses section of the Generics chapter and in the Generic Parameters and Arguments chapter, now that generic `where` clauses are written at the end of a declaration.

- Updated the discussion in the Escaping Closures section, now that closures are nonescaping by default.

- Updated the discussion in the Optional Binding section of the The Basics chapter and the While Statement section of the Statements chapter, now that `if`, `while`, and `guard` statements use a comma-separated list of conditions without `where` clauses.

- Added information about switch cases that have multiple patterns to the Switch section of the Control Flow chapter and the Switch Statement section of the Statements chapter.

- Updated the discussion of function types in the Function Type section now that function argument labels are no longer part of a function's type.
- Updated the discussion of protocol composition types in the Protocol Composition section of the Protocols chapter and in the Protocol Composition Type section of the Types chapter to use the new `Protocol1 & Protocol2` syntax.
- Updated the discussion in the Dynamic Type Expression section to use the new `type(of:)` syntax for dynamic type expressions.
- Updated the discussion of line control statements to use the `#sourceLocation(file:line:)` syntax in the Line Control Statement section.
- Updated the discussion in Functions that Never Return to use the new `Never` type.
- Added information about playground literals to the Literal Expression section.
- Updated the discussion in the In-Out Parameters section to note that only nonescaping closures can capture in-out parameters.
- Updated the discussion about default parameters in the Default Parameter Values section, now that they can't be reordered in function calls.
- Updated attribute arguments to use a colon in the Attributes chapter.
- Added information about throwing an error inside the catch block of a rethrowing function to the Rethrowing Functions and Methods section.
- Added information about accessing the selector of an Objective-C property's getter or setter to the Selector Expression section.

- Added information to the Type Alias Declaration section about generic type aliases and using type aliases inside of protocols.
- Updated the discussion of function types in the Function Type section to note that parentheses around the parameter types are required.
- Updated the Attributes chapter to note that the `@IBAction`, `@IBOutlet`, and `@NSManaged` attributes imply the `@objc` attribute.
- Added information about the `@GKInspectable` attribute to the Declaration Attributes section.
- Updated the discussion of optional protocol requirements in the Optional Protocol Requirements section to clarify that they're used only in code that interoperates with Objective-C.
- Removed the discussion of explicitly using `let` in function parameters from the Function Declaration section.
- Removed the discussion of the `Boolean` protocol from the Statements chapter, now that the protocol has been removed from the Swift standard library.
- Corrected the discussion of the `@NSApplicationMain` attribute in the Declaration Attributes section.

2016-03-21

- Updated for Swift 2.2.
- Added information about how to conditionally compile code depending on the version of Swift being used to the Conditional Compilation Block section.
- Added information about how to distinguish between methods or initializers whose names differ only by the names of their arguments to the Explicit Member Expression section.

- Added information about the `#selector` syntax for Objective-C selectors to the Selector Expression section.
- Updated the discussion of associated types to use the `associatedtype` keyword in the Associated Types and Protocol Associated Type Declaration sections.
- Updated information about initializers that return `nil` before the instance is fully initialized in the Failable Initializers section.
- Added information about comparing tuples to the Comparison Operators section.
- Added information about using keywords as external parameter names to the Keywords and Punctuation section.
- Updated the discussion of the `@objc` attribute in the Declaration Attributes section to note that enumerations and enumeration cases can use this attribute.
- Updated the Operators section with discussion of custom operators that contain a dot.
- Added a note to the Rethrowing Functions and Methods section that rethrowing functions can't directly throw errors.
- Added a note to the Property Observers section about property observers being called when you pass a property as an in-out parameter.
- Added a section about error handling to the A Swift Tour chapter.
- Updated figures in the Weak References section to show the deallocation process more clearly.
- Removed discussion of C-style `for` loops, the `++` prefix and postfix operators, and the `--` prefix and postfix operators.

- Removed discussion of variable function arguments and the special syntax for curried functions.

2015-10-20

- Updated for Swift 2.1.
- Updated the String Interpolation and String Literals sections now that string interpolations can contain string literals.
- Added the Escaping Closures section with information about the `@noescape` attribute.
- Updated the Declaration Attributes and Conditional Compilation Block sections with information about tvOS.
- Added information about the behavior of in-out parameters to the In-Out Parameters section.
- Added information to the Capture Lists section about how values specified in closure capture lists are captured.
- Updated the Accessing Properties Through Optional Chaining section to clarify how assignment through optional chaining behaves.
- Improved the discussion of autoclosures in the Autoclosures section.
- Added an example that uses the `??` operator to the A Swift Tour chapter.

2015-09-16

- Updated for Swift 2.0.
- Added information about error handling to the Error Handling chapter, the Do Statement section, the Throw Statement section, the Defer Statement section, and the Try Operator section.

- Updated the Representing and Throwing Errors section, now that all types can conform to the `ErrorType` protocol.
- Added information about the new `try?` keyword to the Converting Errors to Optional Values section.
- Added information about recursive enumerations to the Recursive Enumerations section of the Enumerations chapter and the Enumerations with Cases of Any Type section of the Declarations chapter.
- Added information about API availability checking to the Checking API Availability section of the Control Flow chapter and the Availability Condition section of the Statements chapter.
- Added information about the new `guard` statement to the Early Exit section of the Control Flow chapter and the Guard Statement section of the Statements chapter.
- Added information about protocol extensions to the Protocol Extensions section of the Protocols chapter.
- Added information about access control for unit testing to the Access Levels for Unit Test Targets section of the Access Control chapter.
- Added information about the new optional pattern to the Optional Pattern section of the Patterns chapter.
- Updated the Repeat-While section with information about the `repeat-while` loop.
- Updated the Strings and Characters chapter, now that `String` no longer conforms to the `CollectionType` protocol from the Swift standard library.
- Added information about the new Swift standard library `print(_:separator:terminator)` function to the Printing Constants

and Variables section.

- Added information about the behavior of enumeration cases with `String` raw values to the Implicitly Assigned Raw Values section of the Enumerations chapter and the Enumerations with Cases of a Raw-Value Type section of the Declarations chapter.

- Added information about the `@autoclosure` attribute—including its `@autoclosure(escaping)` form—to the Autoclosures section.

- Updated the Declaration Attributes section with information about the `@available` and `@warn_unused_result` attributes.

- Updated the Type Attributes section with information about the `@convention` attribute.

- Added an example of using multiple optional bindings with a `where` clause to the Optional Binding section.

- Added information to the String Literals section about how concatenating string literals using the `+` operator happens at compile time.

- Added information to the Metatype Type section about comparing metatype values and using them to construct instances with initializer expressions.

- Added a note to the Debugging with Assertions section about when user-defined assertions are disabled.

- Updated the discussion of the `@NSManaged` attribute in the Declaration Attributes section, now that the attribute can be applied to certain instance methods.

- Updated the Variadic Parameters section, now that variadic parameters can be declared in any position in a function's parameter list.

- Added information to the Overriding a Failable Initializer section about how a nonfailable initializer can delegate up to a failable initializer by force-unwrapping the result of the superclass's initializer.
- Added information about using enumeration cases as functions to the Enumerations with Cases of Any Type section.
- Added information about explicitly referencing an initializer to the Initializer Expression section.
- Added information about build configuration and line control statements to the Compiler Control Statements section.
- Added a note to the Metatype Type section about constructing class instances from metatype values.
- Added a note to the Weak References section about weak references being unsuitable for caching.
- Updated a note in the Type Properties section to mention that stored type properties are lazily initialized.
- Updated the Capturing Values section to clarify how variables and constants are captured in closures.
- Updated the Declaration Attributes section to describe when you can apply the `@objc` attribute to classes.
- Added a note to the Handling Errors section about the performance of executing a `throw` statement. Added similar information about the `do` statement in the Do Statement section.
- Updated the Type Properties section with information about stored and computed type properties for classes, structures, and enumerations.
- Updated the Break Statement section with information about labeled break statements.

- Updated a note in the Property Observers section to clarify the behavior of `willSet` and `didSet` observers.
- Added a note to the Access Levels section with information about the scope of `private` access.
- Added a note to the Weak References section about the differences in weak references between garbage collected systems and ARC.
- Updated the Special Characters in String Literals section with a more precise definition of Unicode scalars.

2015-04-08

- Updated for Swift 1.2.
- Swift now has a native `Set` collection type. For more information, see Sets.
- `@autoclosure` is now an attribute of the parameter declaration, not its type. There's also a new `@noescape` parameter declaration attribute. For more information, see Declaration Attributes.
- Type methods and properties now use the `static` keyword as a declaration modifier. For more information see Type Variable Properties.
- Swift now includes the `as?` and `as!` failable downcast operators. For more information, see Checking for Protocol Conformance.
- Added a new guide section about String Indices.
- Removed the overflow division (`&/`) and overflow remainder (`&%`) operators from Overflow Operators.
- Updated the rules for constant and constant property declaration and initialization. For more information, see Constant Declaration.

- Updated the definition of Unicode scalars in string literals. See Special Characters in String Literals.
- Updated Range Operators to note that a half-open range with the same start and end index will be empty.
- Updated Closures Are Reference Types to clarify the capturing rules for variables.
- Updated Value Overflow to clarify the overflow behavior of signed and unsigned integers
- Updated Protocol Declaration to clarify protocol declaration scope and members.
- Updated Defining a Capture List to clarify the syntax for weak and unowned references in closure capture lists.
- Updated Operators to explicitly mention examples of supported characters for custom operators, such as those in the Mathematical Operators, Miscellaneous Symbols, and Dingbats Unicode blocks.
- Constants can now be declared without being initialized in local function scope. They must have a set value before first use. For more information, see Constant Declaration.
- In an initializer, constant properties can now only assign a value once. For more information, see Assigning Constant Properties During Initialization.
- Multiple optional bindings can now appear in a single `if` statement as a comma-separated list of assignment expressions. For more information, see Optional Binding.
- An Optional-Chaining Expression must appear within a postfix expression.

- Protocol casts are no longer limited to `@objc` protocols.
- Type casts that can fail at runtime now use the `as?` or `as!` operator, and type casts that are guaranteed not to fail use the `as` operator. For more information, see Type-Casting Operators.

2014-10-16

- Updated for Swift 1.1.
- Added a full guide to Failable Initializers.
- Added a description of Failable Initializer Requirements for protocols.
- Constants and variables of type `Any` can now contain function instances. Updated the example in Type Casting for Any and AnyObject to show how to check for and cast to a function type within a `switch` statement.
- Enumerations with raw values now have a `rawValue` property rather than a `toRaw()` method and a failable initializer with a `rawValue` parameter rather than a `fromRaw()` method. For more information, see Raw Values and Enumerations with Cases of a Raw-Value Type.
- Added a new reference section about Failable Initializers, which can trigger initialization failure.
- Custom operators can now contain the `?` character. Updated the Operators reference to describe the revised rules. Removed a duplicate description of the valid set of operator characters from Custom Operators.

2014-08-18

- New document that describes Swift 1.0, Apple's new programming language for building iOS and OS X apps.
- Added a new section about Initializer Requirements in protocols.

- Added a new section about Class-Only Protocols.

- Assertions and Preconditions can now use string interpolation. Removed a note to the contrary.

- Updated the Concatenating Strings and Characters section to reflect the fact that `String` and `Character` values can no longer be combined with the addition operator (`+`) or addition assignment operator (`+=`). These operators are now used only with `String` values. Use the `String` type's `append(_:)` method to append a single `Character` value onto the end of a string.

- Added information about the `availability` attribute to the Declaration Attributes section.

- Optionals no longer implicitly evaluate to `true` when they have a value and `false` when they do not, to avoid confusion when working with optional `Bool` values. Instead, make an explicit check against `nil` with the `==` or `!=` operators to find out if an optional contains a value.

- Swift now has a Nil-Coalescing Operator (`a ?? b`), which unwraps an optional's value if it exists, or returns a default value if the optional is `nil`.

- Updated and expanded the Comparing Strings section to reflect and demonstrate that string and character comparison and prefix / suffix comparison are now based on Unicode canonical equivalence of extended grapheme clusters.

- You can now try to set a property's value, assign to a subscript, or call a mutating method or operator through Optional Chaining. The information about Accessing Properties Through Optional Chaining has been updated accordingly, and the examples of checking for method call success in Calling Methods Through Optional Chaining have been expanded to show how to check for property setting success.

- Added a new section about Accessing Subscripts of Optional Type through optional chaining.

- Updated the Accessing and Modifying an Array section to note that you can no longer append a single item to an array with the `+=` operator. Instead, use the `append(_:)` method, or append a single-item array with the `+=` operator.

- Added a note that the start value `a` for the Range Operators `a...b` and `a..<b` must not be greater than the end value `b`.

- Rewrote the Inheritance chapter to remove its introductory coverage of initializer overrides. This chapter now focuses more on the addition of new functionality in a subclass, and the modification of existing functionality with overrides. The chapter's example of Overriding Property Getters and Setters has been rewritten to show how to override a `description` property. (The examples of modifying an inherited property's default value in a subclass initializer have been moved to the Initialization chapter.)

- Updated the Initializer Inheritance and Overriding section to note that overrides of a designated initializer must now be marked with the `override` modifier.

- Updated the Required Initializers section to note that the `required` modifier is now written before every subclass implementation of a required initializer, and that the requirements for required initializers can now be satisfied by automatically inherited initializers.

- Infix Operator Methods no longer require the `@infix` attribute.

- The `@prefix` and `@postfix` attributes for Prefix and Postfix Operators have been replaced by `prefix` and `postfix` declaration modifiers.

- Added a note about the order in which Prefix and Postfix Operators are applied when both a prefix and a postfix operator are applied to the same operand.

- Operator functions for Compound Assignment Operators no longer use the `@assignment` attribute when defining the function.
- The order in which modifiers are specified when defining Custom Operators has changed. You now write `prefix operator` rather than `operator prefix`, for example.
- Added information about the `dynamic` declaration modifier in Declaration Modifiers.
- Added information about how type inference works with Literals.
- Added more information about curried functions.
- Added a new chapter about Access Control.
- Updated the Strings and Characters chapter to reflect the fact that Swift's `Character` type now represents a single Unicode extended grapheme cluster. Includes a new section on Extended Grapheme Clusters and more information about Unicode Scalar Values and Comparing Strings.
- Updated the String Literals section to note that Unicode scalars inside string literals are now written as `\u{n}`, where `n` is a hexadecimal number between 0 and 10FFFF, the range of Unicode's codespace.
- The `NSString length` property is now mapped onto Swift's native `String` type as `utf16Count`, not `utf16count`.
- Swift's native `String` type no longer has an `uppercaseString` or `lowercaseString` property. The corresponding section in Strings and Characters has been removed, and various code examples have been updated.
- Added a new section about Initializer Parameters Without Argument Labels.

- Added a new section about Required Initializers.
- Added a new section about Optional Tuple Return Types.
- Updated the Type Annotations section to note that multiple related variables can be defined on a single line with one type annotation.
- The `@optional`, `@lazy`, `@final`, and `@required` attributes are now the `optional`, `lazy`, `final`, and `required` Declaration Modifiers.
- Updated the entire book to refer to `..<` as the Half-Open Range Operator (rather than the "half-closed range operator").
- Updated the Accessing and Modifying a Dictionary section to note that `Dictionary` now has a Boolean `isEmpty` property.
- Clarified the full list of characters that can be used when defining Custom Operators.
- `nil` and the Booleans `true` and `false` are now Literals.
- Swift's `Array` type now has full value semantics. Updated the information about Mutability of Collections and Arrays to reflect the new approach. Also clarified the assignment and copy behavior for strings arrays and dictionaries.
- Array Type Shorthand Syntax is now written as `[SomeType]` rather than `SomeType[]`.
- Added a new section about Dictionary Type Shorthand Syntax, which is written as `[KeyType: ValueType]`.
- Added a new section about Hash Values for Set Types.
- Examples of Closure Expressions now use the global `sorted(_:_:)` function rather than the global `sort(_:_:)` function, to reflect the new array value semantics.

- Updated the information about Memberwise Initializers for Structure Types to clarify that the memberwise structure initializer is made available even if a structure's stored properties don't have default values.
- Updated to `..<` rather than `..` for the Half-Open Range Operator.
- Added an example of Extending a Generic Type.

Copyright and Notices

Apple Inc.
Copyright © 2021 Apple Inc.

This document is made available under a Creative Commons Attribution 4.0 International (CC BY 4.0) License:
https://creativecommons.org/licenses/by/4.0/

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document.

Apple Inc.
One Apple Park Way
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Bonjour, Cocoa, Logic, Mac, Numbers, Objective-C, OS X, Retina, Sand, Shake, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

Swift and tvOS are trademarks of Apple Inc.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Converted by Evan at Apps Dissected -www.appsdissected.com