About Swift

Swift is a new programming language for iOS, macOS, watchOS, and tvOS apps that builds on the best of C and Objective-C, without the constraints of C compatibility. Swift adopts safe programming patterns and adds modern features to make programming easier, more flexible, and more fun. Swift’s clean slate, backed by the mature and much-loved Cocoa and Cocoa Touch frameworks, is an opportunity to reimagine how software development works.

Swift has been years in the making. Apple laid the foundation for Swift by advancing our existing compiler, debugger, and framework infrastructure. We simplified memory management with Automatic Reference Counting (ARC). Our framework stack, built on the solid base of Foundation and Cocoa, has been modernized and standardized throughout. Objective-C itself has evolved to support blocks, collection literals, and modules, enabling framework adoption of modern language technologies without disruption. Thanks to this groundwork, we can now introduce a new language for the future of Apple software development.

Swift feels familiar to Objective-C developers. It adopts the readability of Objective-C’s named parameters and the power of Objective-C’s dynamic object model. It provides seamless access to existing Cocoa frameworks and mix-and-match interoperability with Objective-C code. Building from this common ground, Swift introduces many new features and unifies the procedural and object-oriented portions of the language.

Swift is friendly to new programmers. It is the first industrial-quality systems programming language that is as expressive and enjoyable as a scripting language. It supports playgrounds, an innovative feature that allows programmers to experiment with Swift code and see the results immediately, without the overhead of building and running an app.

Swift combines the best in modern language thinking with wisdom from the wider Apple engineering culture. The compiler is optimized for performance, and the language is optimized for development, without compromising on either. It’s designed to scale from “hello, world” to an entire operating system. All this makes Swift a sound future investment for developers and for Apple.

Swift is a fantastic way to write iOS, macOS, watchOS, and tvOS apps, and will continue to evolve with new features and capabilities. Our goals for Swift are ambitious. We can’t wait to see what you create with it.

A Swift Tour

按照学习新语言的惯例，第一个程序是在屏幕上打印单词“Hello, world!”。在Swift, 一行代码就可以完成这个:

* print("Hello, world!")

如果你有过C语言或者OC语言的经历, 你就会很熟悉这个语法—在Swift, 这一行代码就是一个完整的程序. 你不需要为输入/输出（input/output）或者字符串处理（string handling）功能导入独立的库（library）。全局范围内编写的代码是程序的入口点, 因此你不需要一个 main() 函数. 你也不需要在每一个语句结束的时候写分号。

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

For the best experience, open this chapter as a playground in Xcode. Playgrounds allow you to edit the code listings and see the result immediately.

[**Download Playground**](https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/GuidedTour.playground.zip)

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 there is no initial value), specify the type by writing it after the variable, separated by a colon.

* let implicitInteger = 70
* let implicitDouble = 70.0
* let explicitDouble: a href="" Double /a = 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.

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", "blue paint"]
* shoppingList[1] = "bottle of water"
* var occupations = [
* "Malcolm": "Captain",
* "Kaylee": "Mechanic",
* ]
* occupations["Jayne"] = "Public Relations"

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, for, 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)

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: a href="" String /a ? = "Hello"
* print(optionalString == nil)
* var optionalName: a href="" String /a ? = "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: a href="" String /a ? = nil
* let fullName: a href="" String /a = "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.")
* }

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 there is no 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 (kind, numbers) in interestingNumbers {
* for number in numbers {
* if number > largest {
* largest = number
* }
* }
* }
* print(largest)

EXPERIMENT

Add another variable to keep track of which kind of number was the largest, as well as what that largest number was.

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 = n \* 2
* }
* print(n)
* var m = 2
* repeat {
* m = m \* 2
* } while m < 100
* print(m)

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)

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: a href="" String /a , day: a href="" String /a ) -> a href="" String /a {
* 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: a href="" String /a , on day: a href="" String /a ) -> a href="" String /a {
* 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: [ a href="" Int /a ]) -> (min: a href="" Int /a , max: a href="" Int /a , sum: a href="" Int /a ) {
* 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)
* print(statistics.2)

Functions can also take a variable number of arguments, collecting them into an array.

* func sumOf(numbers: a href="" Int /a ...) -> a href="" Int /a {
* var sum = 0
* for number in numbers {
* sum += number
* }
* return sum
* }
* sumOf()
* sumOf(numbers: 42, 597, 12)

EXPERIMENT

Write a function that calculates the average of its arguments.

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 is long or complex.

* func returnFifteen() -> a href="" Int /a {
* 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() -> (( a href="" Int /a ) -> a href="" Int /a ) {
* func addOne(number: a href="" Int /a ) -> a href="" Int /a {
* return 1 + number
* }
* return addOne
* }
* var increment = makeIncrementer()
* increment(7)

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

* func hasAnyMatches(list: [ a href="" Int /a ], condition: ( a href="" Int /a ) -> a href="" Bool /a ) -> a href="" Bool /a {
* for item in list {
* if condition(item) {
* return true
* }
* }
* return false
* }
* func lessThanTen(number: a href="" Int /a ) -> a href="" Bool /a {
* 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 is 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: a href="" Int /a ) -> a href="" Int /a 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)

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)

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 is in the context of a class. Likewise, method and function declarations are written the same way.

* class Shape {
* var numberOfSides = 0
* func simpleDescription() -> a href="" String /a {
* 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: a href="" Int /a = 0
* var name: a href="" String /a
* init(name: a href="" String /a ) {
* self.name = name
* }
* func simpleDescription() -> a href="" String /a {
* 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 is 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: a href="" NamedShape /a {
* var sideLength: a href="" Double /a
* init(sideLength: a href="" Double /a , name: a href="" String /a ) {
* self.sideLength = sideLength
* super.init(name: name)
* numberOfSides = 4
* }
* func area() -> a href="" Double /a {
* return sideLength \* sideLength
* }
* override func simpleDescription() -> a href="" String /a {
* 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: a href="" NamedShape /a {
* var sideLength: a href="" Double /a = 0.0
* init(sideLength: a href="" Double /a , name: a href="" String /a ) {
* self.sideLength = sideLength
* super.init(name: name)
* numberOfSides = 3
* }
* var perimeter: a href="" Double /a {
* get {
* return 3.0 \* sideLength
* }
* set {
* sideLength = newValue / 3.0
* }
* }
* override func simpleDescription() -> a href="" String /a {
* return "An equilateral triangle with sides of length \(sideLength)."
* }
* }
* var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
* print(triangle.perimeter)
* triangle.perimeter = 9.9
* print(triangle.sideLength)

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 is 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: a href="" EquilateralTriangle /a {
* willSet {
* square.sideLength = newValue.sideLength
* }
* }
* var square: a href="" Square /a {
* willSet {
* triangle.sideLength = newValue.sideLength
* }
* }
* init(size: a href="" Double /a , name: a href="" String /a ) {
* 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)
* print(triangleAndSquare.triangle.sideLength)
* triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
* print(triangleAndSquare.triangle.sideLength)

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: a href="" Square /a ? = 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: a href="" Int /a {
* case ace = 1
* case two, three, four, five, six, seven, eight, nine, ten
* case jack, queen, king
* func simpleDescription() -> a href="" String /a {
* 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.

* 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() -> a href="" String /a {
* 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.

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 are passed around in your code, but classes are passed by reference.

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

EXPERIMENT

Add a method to Card that creates a full deck of cards, with one card of each combination of rank and suit.

An instance of an enumeration case can have values associated with the instance. Instances of the same enumeration case can have different values associated with them. You provide the associated values when you create the instance. Associated values and raw values are different: The raw value of an enumeration case is the same for all of its instances, and you provide the raw value when you define the enumeration.

For example, consider the case of requesting the sunrise and sunset time from a server. The server either responds with the information or it responds with some error information.

* 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)")
* }

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.

Protocols and Extensions

Use protocol to declare a protocol.

* protocol ExampleProtocol {
* var simpleDescription: a href="" String /a { get }
* mutating func adjust()
* }

Classes, enumerations, and structs can all adopt protocols.

* class SimpleClass: a href="" ExampleProtocol /a {
* var simpleDescription: a href="" String /a = "A very simple class."
* var anotherProperty: a href="" Int /a = 69105
* func adjust() {
* simpleDescription += " Now 100% adjusted."
* }
* }
* var a = SimpleClass()
* a.adjust()
* let aDescription = a.simpleDescription
* struct SimpleStructure: a href="" ExampleProtocol /a {
* var simpleDescription: a href="" String /a = "A simple structure"
* mutating func adjust() {
* simpleDescription += " (adjusted)"
* }
* }
* var b = SimpleStructure()
* b.adjust()
* let bDescription = b.simpleDescription

EXPERIMENT

Write an enumeration that conforms to this 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 is declared elsewhere, or even to a type that you imported from a library or framework.

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

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 are not available.

* let protocolValue: a href="" ExampleProtocol /a = a
* print(protocolValue.simpleDescription)
* // 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: a href="" Error /a {
* 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: a href="" Int /a , toPrinter printerName: a href="" String /a ) throws -> a href="" String /a {
* 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)
* }

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 a href="" PrinterError /a {
* print("Printer error: \(printerError).")
* } catch {
* print(error)
* }

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 is 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: a href="" String /a ) -> a href="" Bool /a {
* fridgeIsOpen = true
* defer {
* fridgeIsOpen = false
* }
* let result = fridgeContent.contains(food)
* return result
* }
* fridgeContains("banana")
* print(fridgeIsOpen)

Generics

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

* func makeArray<Item>(repeating item: a href="" Item /a , numberOfTimes: a href="" Int /a ) -> [ a href="" Item /a ] {
* var result = [ a href="" Item /a ]()
* 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.

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

* func anyCommonElements<T: a href="" Sequence /a , U: a href="" Sequence /a >(\_ lhs: T, \_ rhs: U) -> Bool
* where a href="" T /a . a href="" Iterator /a . a href="" Element /a : a href="" Equatable /a , a href="" T /a . a href="" Iterator /a . a href="" Element /a == a href="" U /a . a href="" Iterator /a . a href="" Element /a {
* 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>.