Swift Course

Flavio Copes

Updated August 3, 2026

Contents

Learn Swift fundamentals through values, optionals, control flow, collections, functions, structures, enumerations, classes, and protocols.

What you'll learn: Write small Swift programs and model data deliberately with values, collections, functions, structures, enumerations, classes, and protocols.

Start with Swift

Understand the language, constants, variables, Booleans, and number types.

Introduction to the Swift programming language

An introduction to Swift, the statically typed, compiled language Apple created in 2014 to build apps for iOS, iPadOS, watchOS, macOS, and beyond.

> This post is the beginning of a new series on Swift

Swift is a general-purpose programming language created by Apple and introduced in 2014.

It is the main language used to build new applications across Apple platforms. It also runs on Linux and Windows, and people use it for servers, command-line tools, and cross-platform packages.

Swift is open source, compiled, and statically typed.

This is a complete Swift program:

```
let name = "Flavio"
print("Hello \(name)")
```

The Swift language guide is the authoritative reference for the language.

Swift is statically typed

Every value in Swift has a type. The compiler checks those types before it builds the program.

You can write the type explicitly:

```
let score: Int = 10
```

Swift can often infer it:

```
let score = 10
```

In both examples, `score` is an `Int`.

Swift is also type-safe. For example, you cannot pass a `String` to a function that requires an `Int`.

Optionals make the absence of a value explicit:

```
var nickname: String? = nil
```

The `?` tells us that `nickname` can contain a string or no value. Swift then makes us handle both cases safely.

See The Basics for the official explanation of types, constants, variables, and optionals.

Where can you use Swift?

Swift is closely associated with Apple platforms, but it is not limited to them.

The official Swift platform support page lists macOS, Linux distributions, and Windows as development platforms. A Swift toolchain includes the compiler, standard library, Swift Package Manager, and development tools for the supported platform.

On a Mac, Xcode includes everything needed to build applications for Apple platforms. Xcode also gives you a visual interface builder, simulator, debugger, and code completion.

Swift Package Manager handles dependencies and builds Swift packages. It is part of the Swift toolchain, not an extra package manager you have to install separately.

How to start learning Swift

If you use a Mac or iPad, Swift Playground is a friendly place to begin. You can run small pieces of Swift without creating a complete application first.

On a Mac, install Xcode when you are ready to build an app.

On Linux or Windows, follow the official Swift getting started guide to install the correct toolchain.

After installing a toolchain, create a small executable package:

```
mkdir HelloSwift
cd HelloSwift
swift package init --type executable
swift run
```

This creates a Swift Package Manager project and runs it.

The goal of this series is to get you comfortable with Swift from zero. Continue with:

- Variables
- Objects
- Basic Operators
- `if` conditionals
- `switch` conditionals
- Ternary conditionals
- `for-in` loops
- `while` loops
- `repeat-while` loops
- Loop control transfer statements
- Comments
- Semicolons
- Numbers
- Strings
- Booleans
- Arrays
- Sets
- Dictionaries
- Tuples
- nil and Optionals
- Enums
- Structs
- Classes
- Functions
- Protocols

Swift Variables

Learn how variables work in Swift, the difference between var and let constants, how type inference assigns a type, and why a variable is bound to it.

This tutorial belongs to the Swift series

Variables let us assign a value to a label, and are defined using the `var` keyword:

```
var name = "Roger"
var age = 8
```

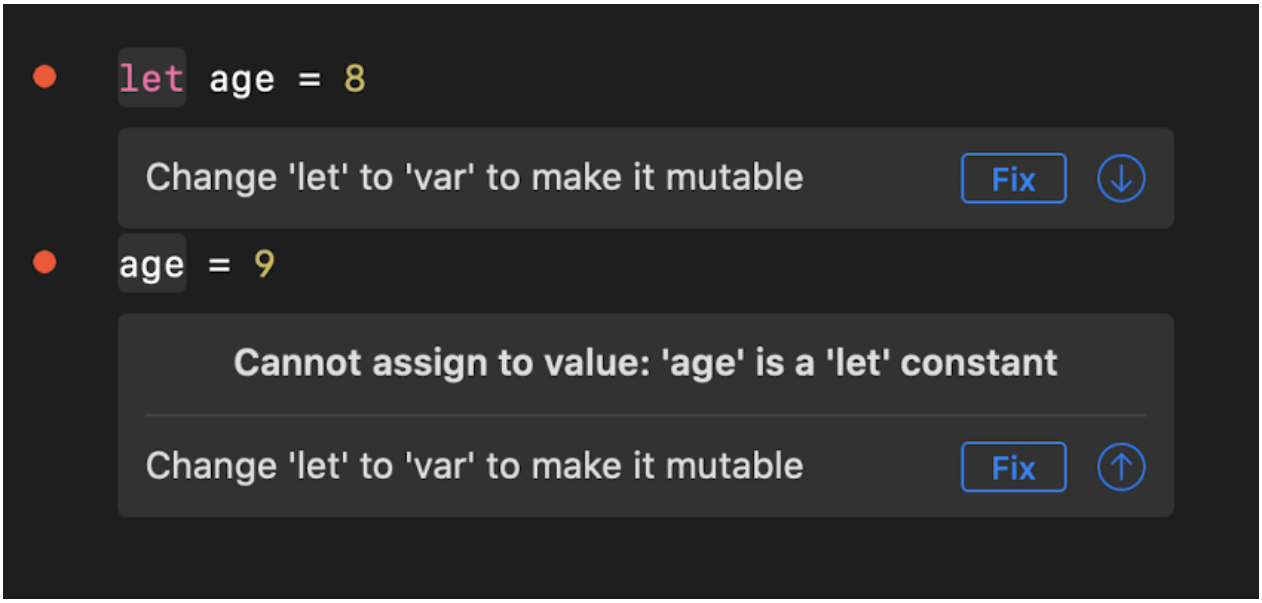
Once a variable is defined, we can change its value:

```
age = 9
```

Variables that you do not want to change can be defined as constants, using the `let` keyword:

```
let name = "Roger"
let age = 8
```

Changing the value of a constant is forbidden.



When you define a variable and you assign it a value, Swift implicitly infers the type of it.

`8` is an `Int` value.

`"Roger"` is a `String` value.

A decimal number like `3.14` is a `Double` value.

You can also specify the type at initialization time:

```
let age: Int = 8
```

but it's usual to let Swift infer it, and it's mostly done when you declare a variable without initializing it.

You can declare a constant, and initialize it later:

```
let age : Int

age = 8
```

Once a variable is defined, it is bound to that type, and you cannot assign to it a different type, unless you explicitly convert it.

You can't do this:

```
var age = 8
age = "nine"
```

`Int` and `String` are just two of the built-in data types provided by Swift.

Booleans in Swift

Learn booleans in Swift: the Bool type, comparison operators, combining conditions with logical operators, toggle, and why Swift has no truthiness.

This tutorial belongs to the Swift series

Swift provides the **`Bool`** type to represent truth values. A `Bool` can only hold two values: `true` or `false`.

You declare a boolean like any other variable:

```
var done = false
```

Since we declared it with `var`, we can reassign it later:

```
var done = false
done = true
```

You can also write the type explicitly:

```
var done: Bool = false
```

Where do booleans come from?

Most of the time you don't type `true` or `false` by hand. You get booleans as the result of **comparison operators**:

```
let age = 18

age == 18 // true
age != 21 // true
age > 16  // true
age < 10  // false
```

Each of those expressions produces a `Bool`, and you can store it in a constant with a meaningful name:

```
let canVote = age >= 18 // true
```

My advice is to name booleans like a question with a yes/no answer: `canVote`, `isLoggedIn`, `hasTicket`. The code that uses them reads much better.

Combining booleans

Swift gives you three logical operators: `&&` (and), `||` (or), and `!` (not).

```
let isWeekend = true
let isSunny = false

isWeekend && isSunny // false
isWeekend || isSunny // true
!isSunny             // true
```

`&&` is `true` only when both sides are `true`. `||` is `true` when at least one side is. `!` flips the value.

You'll use them all the time in conditionals:

```
let age = 25
let hasTicket = true

if age >= 18 && hasTicket {
    print("Welcome in!")
}
```

Flipping a boolean with toggle()

When you want to invert a boolean, you could write `done = !done`. Swift has a nicer way, the `toggle()` method:

```
var done = false
done.toggle() // done is now true
done.toggle() // done is now false
```

It only works on variables declared with `var`, because it changes the value in place.

Swift has no truthiness

In JavaScript you can write `if (count)` and any non-zero number counts as true. Swift does not allow that. A condition must be a real `Bool`:

```
let count = 3

if count {
    // does not compile
}
```

You have to write the comparison explicitly:

```
let count = 3

if count > 0 {
    print("We have items")
}
```

This feels more verbose at first, but it removes an entire category of bugs. You always say exactly what you're checking.

The = vs == mistake

A classic bug in C-family languages is typing `=` (assignment) when you meant `==` (comparison). The condition assigns a value and the code runs when you didn't expect it to.

Swift catches this at compile time:

```
var logged = false

if logged = true {
    // does not compile
}
```

An assignment does not return a value in Swift, and an `if` condition must be a boolean expression. So the compiler stops you before this mistake ever reaches your users.

Numbers in Swift

Learn how numbers work in Swift, from the main Int and Double types to sized integers like Int8 and UInt, and how to convert one numeric type to another.

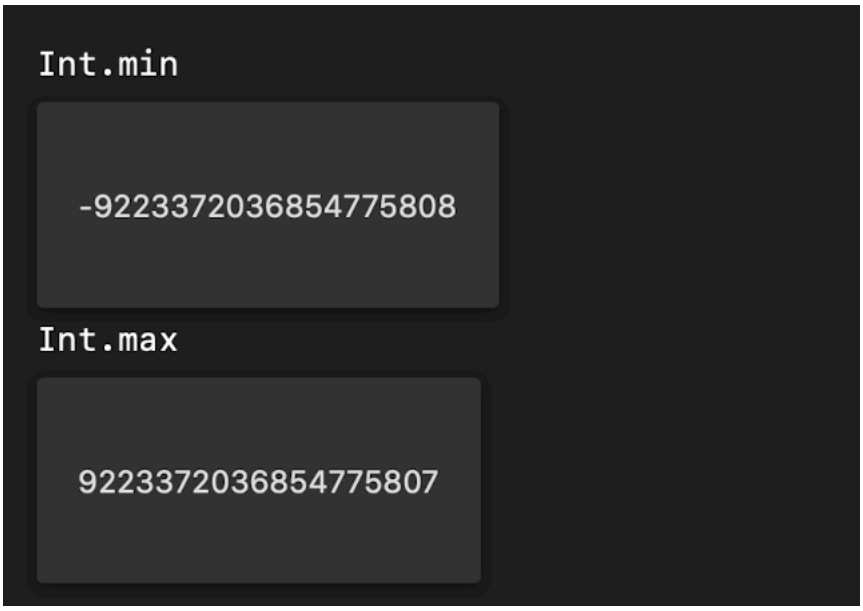
This tutorial belongs to the Swift series

In Swift, numbers have 2 main types: `Int` and `Double`.

An `Int` is a number without decimal point. A `Double` is a number with decimal point.

Both use 64 bits, on modern computers that work with 64 bits, and 32 bit on 32-bit platforms.

The range of values they can store depends on the platform used, and can be retrieved using the `int` property of each type:



Then, in addition to `Int` and `Double`, we have lots of other numeric types, mostly used to interact with APIs built in the past and that needed to interact with C or Objective-C, and you must be aware that we have them:

- `Int8` is an integer with 8 bits
- `Int16` is an integer with 16 bits
- `Int32` is an integer with 32 bits
- `Int64` is an integer with 64 bits
- `UInt8` is an unsigned integer with 8 bits
- `UInt16` is an unsigned integer with 16 bits
- `UInt32` is an unsigned integer with 32 bits
- `UInt64` is an unsigned integer with 64 bits

`UInt` is like `Int`, but unsigned, and it ranges from 0 to `Int.max * 2`.

`Float` is a decimal number with 32 bits.

Then using Cocoa APIs you might use other numeric types like CLong, CGFloat, and more.

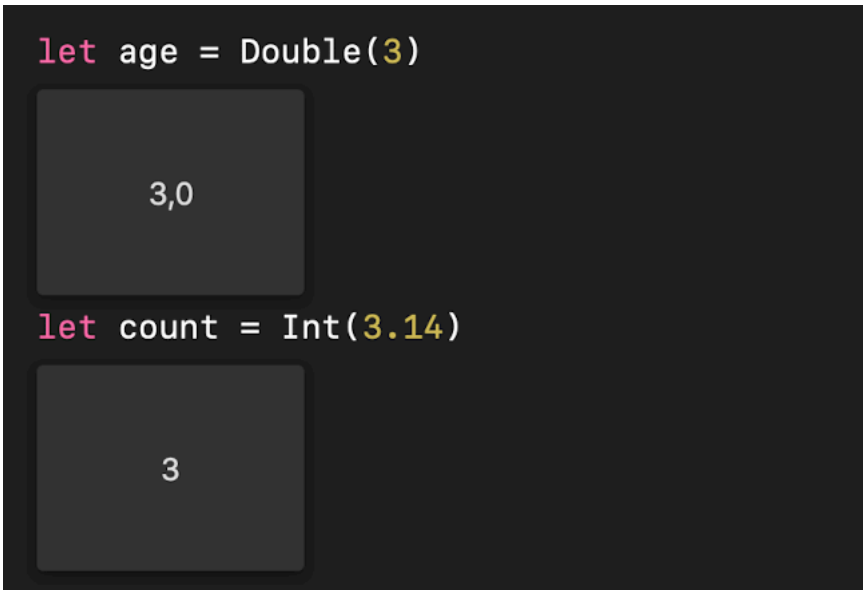
You will always use `Int` or `Double` in your code, and use those specific types to particular cases.

Any of those types can always be converted to `Int` and `Double` types, instantiating a number passing the value inside parentheses to `Double()` or `Int()`:

```
let age : UInt8 = 3
let intAge = Int(age)
```

You can also convert a number from `Double` to `Int`:

```
let age = Double(3)
let count = Int(3.14)
```



Check your understanding: starting with Swift

Check the key ideas from starting with swift before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Values and types

Work with strings, optionals, tuples, assignment, comparison, and other operators.

Strings in Swift

Learn how to work with strings in Swift, including string literals, multi-line strings, interpolation, and concatenation, with beginner-friendly examples.

> This tutorial belongs to the Swift series

Strings are one of the most popular tools in programming.

In Swift, a string can be defined using the string literal syntax:

```
let name = "Roger"
```

We use double quotes. Single quotes are not valid string delimiters.

A string can span over multiple lines, using 3 double quotes:

```
let description = """
    a long
    long
        long description
    """
```

You can use string interpolation to embed an expression in a string:

```
let age = 8

let name = """
    Roger, age \(age)
    Next year he will be \(age + 1)
    """
```

Concatenate two strings with the `+` operator:

```
var name = "Roger"
name = name + " The Dog"
```

Append text to a string with the `+=` operator:

```
var name = "Roger"
name += " The Dog"
```

Or using the `append(_:)` method:

```
var name = "Roger"
name.append(" The Dog")
```

You can count the characters in a string using the `count` string property:

```
let name = "Roger"
name.count //5
```

Any string comes with a set of useful methods, for example:

- `removeFirst()` to remove the first character
- `removeLast()` to remove the last character
- `lowercased()` to get a new string, lowercased
- `uppercased()` to get a new string, uppercased
- `starts(with:)` which returns true if the string starts with a specific substring
- `contains()` which returns true if the string contains a specific character

and many, many more.

When you need to reference an item into the string, since strings in Swift are unicode, we can't simply reference the letter `o` in `let name = "Roger"` using `name[1]`. You need to work with indexes.

Any string provides the starting index with the `startIndex` property:

```
let name = "Roger"
name.startIndex //0
```

To calculate a specific index in the string, you calculate it using the `index(i:offsetBy:)` method:

```
let name = "Roger"
let i = name.index(name.startIndex, offsetBy: 2)
name[i] //"g"
```

The index can be used also to get a substring:

```
let name = "Roger"
let i = name.index(name.startIndex, offsetBy: 2)
name.suffix(from: i) //"ger"

//Or using the subscript:

name[i...] //"ger"
```

When you get a substring from a string, the type of the result is `Substring`, not `String`.

```
let name = "Roger"
let i = name.index(name.startIndex, offsetBy: 2)
print(type(of: name.suffix(from: i)))
//Substring
```

Substrings are more memory efficient, because you do not get a new string, but the same memory structure is used behind the scenes, although you need to be careful when you deal with strings a lot, as there are optimizations you can implement.

Strings are collections, and they can be iterated over in loops.

Swift optionals and nil

Learn how optionals and nil work in Swift, how to declare an optional with a question mark, and how to safely unwrap its value using an if let statement.

This tutorial belongs to the Swift series

Optionals are one key feature of Swift.

When you don't know if a value will be present or absent, you declare the type as an optional.

The optional wraps another value, with its own type. Or maybe not.

We declare an optional adding a question mark after its type, like this:

```
var value: Int? = 10
```

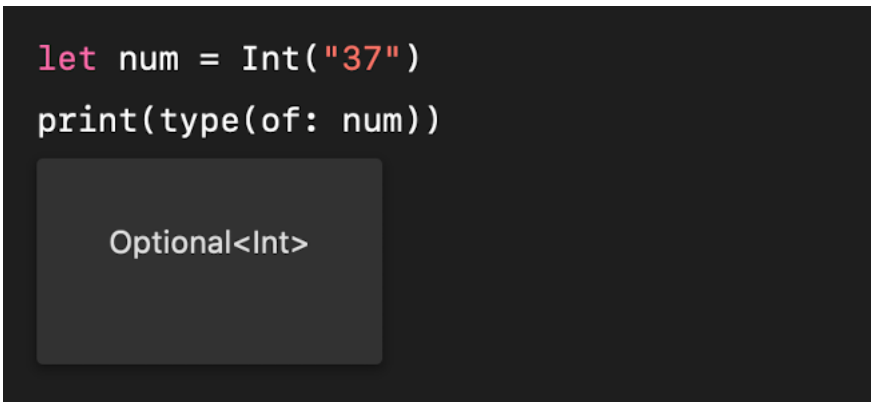
Now value is not an Int value. It's an optional wrapping an Int value.

To find out if the optional wraps a value, you must **unwrap** it.

We do so using an exclamation mark:

```
var value: Int? = 10
print(value!) //10
```

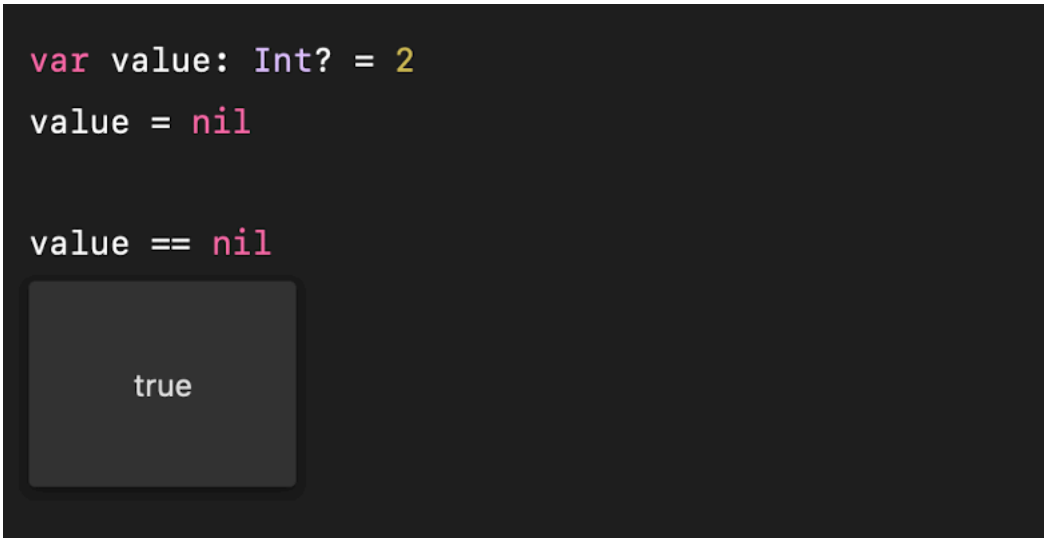
Swift methods often return an optional. For example the `Int` type initializer accepts a string, and returns an Int optional:

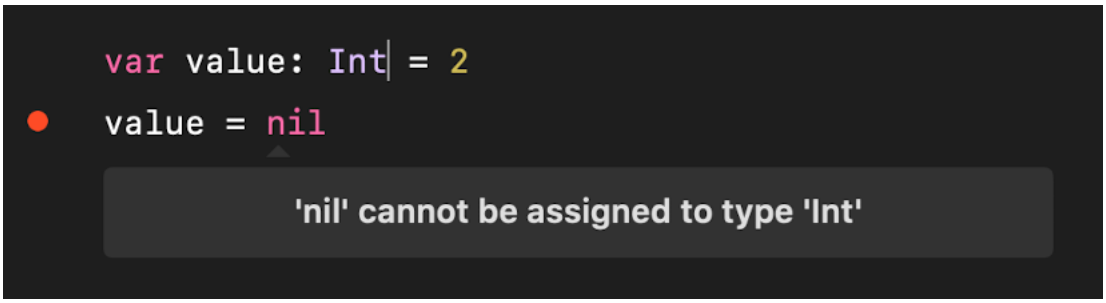


This is because it does not know if the string can be converted to a number.

If the optional does not contain a value, it evaluates as `nil`, and you cannot unwrap it:

`nil` is a special value that cannot be assigned to a variable. Only to an optional:





You typically use `if` statements to unwrap values in your code, like this:

```
var value: Int? = 2

if let age = value {
    print(age)
}
```

Swift Tuples

Learn how to use tuples in Swift to group several values together, name and decompose their elements, return multiple values, and even swap variables.

This tutorial belongs to the Swift series

Tuples are used to group multiple values into a single collection. For example we can declare a variable `dog` containing a String and an Int value:

```
let dog : (String, Int)
```

And we can initialize them with a name and an age

```
let dog : (String, Int) = ("Roger", 8)
```

But as with any other variable, the type can be inferred during initialization:

```
let dog = ("Roger", 8)
```

You can use named elements:

```
let dog = (name: "Roger", age: 8)

dog.name //"Roger"
dog.age //8
```

Once a tuple is defined, you can decompose it to individual variables in this way:

```
let dog = ("Roger", 8)
let (name, age) = dog
```

and if you need to just get one of the values, you can use the special underscore keyword to ignore the other ones:

```
let dog = ("Roger", 8)
let (name, _) = dog
```

Tuples are an awesome tool for various needs.

The most obvious one is a short way to group similar data.

Another one if those needs is returning multiple items from a function. A function can only return a single item, so a tuple is a convenient structure for that.

Another handy functionality allowed by tuples is swapping elements:

```
var a = 1
var b = 2

(a, b) = (b, a)

//a == 2
//b == 1
```

Swift Operators

A guide to Swift operators, covering the assignment, arithmetic, compound assignment, comparison, range, and logical operators you use to work with values.

> This tutorial belongs to the Swift series

We can use a wide set of operators to operate on values.

We can divide operators in many categories. The first is the number of targets: 1 for **unary operators**, 2 for **binary operators** or 3 for the one and only **ternary operator**.

Then we can divide operators based on the kind of operation they perform:

- assignment operator
- arithmetic operators
- compound assignment operators
- comparison operators
- range operators
- logical operators

plus some more advanced ones, including nil-coalescing, ternary conditional, overflow, bitwise and pointwise operators.

> Note: Swift allows you to create your own operators and define how operators work on your types you define.

Assignment operator The assignment operator is used to assign a value to a variable:

```
var age = 8
```

Or to assign a variable value to another variable:

```
var age = 8
var another = age
```

Arithmetic operators Swift has a number of binary arithmetic operators: `+`, `-`, `*`, `/` (division), `%` (remainder):

```
1 + 1 //2
2 - 1 //1
2 * 2 //4
4 / 2 //2
```

```
4 % 3 //1
4 % 2 //0
```

`-` also works as a unary minus operator:

```
let hotTemperature = 20
let freezingTemperature = -20
```

`+` is also used to concatenate String values:

```
"Roger" + " is a good dog"
```

Compound assignment operators The compound assignment operators combine the assignment operator with arithmetic operators:

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`

Example:

```
var age = 8
age += 1
```

Comparison operators Swift defines a few comparison operators:

- `==`
- `!=`
- `>`
- `<`
- `>=`
- `<=`

You can use those operators to get a boolean value (`true` or `false`) depending on the result:

```
let a = 1
let b = 2

a == b //false
a != b //true
a > b // false
a <= b //true
```

Range operators Range operators are used in loops. They allow us to define a range:

```
0...3 //4 times
0..<3 //3 times

0...count //"count" times
0..<count //"count-1" times
```

Here's a sample usage:

```
let count = 3
```

```
for i in 0...count {
  //loop body
}
```

Logical operators Swift gives us the following logical operators:

- `!`, the unary operator NOT
- `&&`, the binary operator AND
- `||`, the binary operator OR

Sample usage:

```
let condition1 = true
let condition2 = false

!condition1 //false

condition1 && condition2 //false
condition1 || condition2 //true
```

Those are mostly used in the `if` conditional expression evaluation:

```
if condition1 && condition2 {
  //if body
}
```

Check your understanding: values and types

Check the key ideas from values and types before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Control flow

Make decisions and repeat work with if, switch, for-in, and while.

Swift conditionals: the if statement

Learn how to use if and else conditionals in Swift, write boolean conditions cleanly, and why Swift prevents the bug of assigning instead of comparing.

This tutorial belongs to the Swift series

`if` statements are the most popular way to perform a conditional check. We use the `if` keyword followed by a boolean expression, followed by a block containing code that is ran if the condition is true:

```
let condition = true
if condition == true {
    // code executed if the condition is true
}
```

An `else` block is executed if the condition is false:

```
let condition = true
if condition == true {
    // code executed if the condition is true
} else {
    // code executed if the condition is false
}
```

You can optionally wrap the condition validation into parentheses if you prefer:

```
if (condition == true) {
    // ...
}
```

And you can also just write:

```
if condition {
    // runs if `condition` is `true`
}
```

or

```
if !condition {
    // runs if `condition` is `false`
}
```

One thing that separates Swift from many other languages is that it prevents bugs caused by erroneously doing an assignment instead of a comparison. This means you can't do this:

```
if condition = true {
    // The program does not compile
}
```

and the reason is that the assignment operator does not return anything, but the `if` conditional must be a boolean expression.

Swift conditionals: the switch statement

Learn the switch statement in Swift: exhaustive cases, no implicit fallthrough, matching ranges, tuples, value binding with case let, and where clauses.

This tutorial belongs to the Swift series

A `switch` statement compares a value against multiple possible cases and runs the code of the first case that matches. It's the cleaner alternative to a long chain of `if`/`else if` blocks:

```
let name = "Roger"

switch name {
case "Roger":
    print("Hello, mr. Roger!")
default:
    print("Hello, \(name)")
}
```

Every case must be covered

Swift requires a `switch` to be **exhaustive**: every possible value of the thing you're switching on must be handled. A string can hold anything, so here the `default` case is mandatory. Remove it and the code doesn't compile.

With an enumeration you can list all the cases instead, and skip `default` entirely:

```
enum Animal {
    case dog
    case cat
}

let animal = Animal.dog

switch animal {
case .dog:
    print("Hello, dog!")
case .cat:
    print("Hello, cat!")
}
```

This is where exhaustiveness earns its keep. Say you later add `case rabbit` to the enum. Every switch that handles `Animal` without a `default` stops compiling, and the compiler points you at each place that needs to handle rabbits. In most languages you'd discover the missing case at runtime, or never.

My advice: when switching on an enum, list the cases explicitly instead of adding a `default`. You give up a little convenience now for a compiler safety net later.

No implicit fallthrough

In C and languages inspired by it, forgetting a `break` makes execution fall into the next case, a classic source of bugs. Swift does the opposite: it runs the matched case and exits the switch. No `break` needed.

If you genuinely want fallthrough behavior, you ask for it with the `fallthrough` keyword:

```
let number = 5

switch number {
case 5:
    print("It's five")
    fallthrough
case 10:
    print("It's five or ten")
default:
    break
}
// prints both lines
```

Note that `fallthrough` does not check the next case's condition, it just executes its body. You'll rarely need it.

That `default: break` is also worth noticing: a case body can't be empty in Swift, so `break` is how you say "do nothing here".

Matching ranges

A case can match a range of values:

```
let age = 20

switch age {
case 0..<18:
    print("You can't drive")
case 18..<70:
    print("You can drive")
default:
    print("Better check with your doctor first")
}
```

Matching tuples

You can switch on a tuple and match its parts individually. The underscore matches any value:

```
let coordinates = (2, 0)

switch coordinates {
case (0, 0):
    print("At the origin")
case (_, 0):
    print("On the x axis")
case (0, _):
    print("On the y axis")
default:
    print("Somewhere else")
}
```

Value binding and where

With `case let` you capture the matched value in a constant, and with `where` you add a condition to the case:

```
let temperature = 34

switch temperature {
case let t where t < 12:
    print("\(t) degrees, bring a jacket")
case let t where t > 30:
    print("\(t) degrees, stay hydrated")
default:
    print("Nice weather")
}
```

The first case whose `where` condition is true wins, and `t` holds the value inside that case's body.

Once you combine ranges, tuples, binding, and `where`, `switch` becomes a small pattern matching engine, far more capable than the switch you may know from other languages.

Swift loops: the for-in loop

Learn the for-in loop in Swift: iterate ranges, arrays, strings and dictionaries, get indexes with enumerated, filter with where, and use stride.

This tutorial belongs to the Swift series

The `for-in` loop is the loop you'll use most in Swift. It iterates over a sequence: a range of numbers, an array, a string, a dictionary. You tell it what to loop over, and Swift hands you one element at a time.

The simplest case is a range. The **closed range** operator `...` includes both ends:

```
for number in 1...5 {
    print(number) // 1, 2, 3, 4, 5
}
```

The **half-open range** operator `..<` excludes the last value:

```
for number in 1..<5 {
    print(number) // 1, 2, 3, 4
}
```

Half-open ranges are handy with array indexes, since an array of 5 items has indexes 0 through 4.

Notice you don't declare `number` with `var` or `let`. The loop creates the constant for you, fresh on each iteration.

Iterating over arrays

Give `for-in` an array and it walks through the elements in order:

```
let fruits = ["apple", "banana", "cherry"]

for fruit in fruits {
    print(fruit)
}
```

What if you also need the index?

Call `enumerated()` on the array. It gives you a tuple with the index and the value:

```
let fruits = ["apple", "banana", "cherry"]

for (index, fruit) in fruits.enumerated() {
    print("\(index): \(fruit)")
}
// 0: apple
// 1: banana
// 2: cherry
```

This is much cleaner than managing a counter variable yourself.

Iterating over strings

A string is a sequence of characters, so you can loop over it directly:

```
for letter in "Swift" {
    print(letter)
}
// S, w, i, f, t
```

Each `letter` is a `Character` value.

Iterating over dictionaries

A dictionary gives you a `(key, value)` tuple on each iteration:

```
let ages = ["Flavio": 40, "Roger": 8, "Syd": 5]

for (name, age) in ages {
    print("\(name) is \(age) years old")
}
```

One thing to remember: dictionaries have no defined order. Run this twice and the lines can come out in a different sequence.

Filtering with where

You can add a `where` clause to skip elements that don't match a condition:

```
for number in 1...10 where number % 2 == 0 {
    print(number) // 2, 4, 6, 8, 10
}
```

This reads better than putting an `if` inside the loop body, because the filter is visible right in the loop declaration.

Custom steps with stride

Ranges always move by 1. When you need a different step, use `stride(from:to:by:)`:

```
for number in stride(from: 0, to: 10, by: 2) {
    print(number) // 0, 2, 4, 6, 8
}
```

The `to:` value is excluded. If you want to include it, use `stride(from:through:by:)` instead:

```
for number in stride(from: 0, through: 10, by: 2) {
    print(number) // 0, 2, 4, 6, 8, 10
}
```

You can also stride backwards with a negative `by:` value.

When you don't need the value

If you just want to repeat something a number of times and don't care about the counter, use an underscore:

```
for _ in 1...3 {
```

```
    print("Hello!")
}
```

The _ tells Swift (and whoever reads your code) that the value is intentionally ignored.

break and continue

Both work inside `for-in` loops. `break` exits the loop, `continue` skips to the next iteration:

```
for number in 1...10 {
    if number == 4 {
        break
    }
    print(number) // 1, 2, 3
}
```

I cover them in more detail in the while loop tutorial, and they behave the same way here.

Swift loops: the while loop

Learn the while and repeat-while loops in Swift: how the condition check works, break and continue, and how to avoid the infinite loop pitfall.

This tutorial belongs to the Swift series

A `while` loop repeats a block of code as long as a condition stays `true`. Swift checks the condition **before** each iteration, so if the condition is false from the start, the body never runs.

Here's the basic form:

```
var count = 0

while count < 3 {
    print(count)
    count += 1
}
// prints 0, 1, 2
```

Follow what happens step by step. `count` starts at 0, so the condition is true and the loop prints 0. Then `count += 1` brings it to 1, still less than 3, so we go again. When `count` reaches 3, the condition becomes false and the loop ends.

And this is a loop that never runs:

```
var count = 10

while count < 3 {
    print("never printed")
}
```

The condition is false at the first check, so Swift skips the body entirely.

What if you need at least one iteration?

Sometimes you want the body to run once no matter what, and check the condition after. Other languages call this a do-while loop. Swift calls it **`repeat-while`**:

```
var count = 10

repeat {
    print(count) // 10
    count += 1
} while count < 3
```

This prints 10 once, even though the condition was already false. The body runs first, then the condition is checked.

`repeat-while` is the right tool when the work itself produces the value you're testing. Think of asking the user for input: you have to ask at least once before you can validate the answer.

A realistic example

A common pattern is processing items until there's nothing left:

```
var tasks = ["email", "meeting", "code review"]

while !tasks.isEmpty {
    let task = tasks.removeFirst()
    print("Doing: \(task)")
}
```

Each iteration removes one task from the array. When the array is empty, `!tasks.isEmpty` is false and the loop stops. The loop condition and the work inside are connected, which is exactly what you want.

break and continue

`break` exits the loop immediately, skipping any remaining iterations:

```
var number = 0

while number < 10 {
    number += 1
    if number == 5 {
        break
    }
    print(number) // 1, 2, 3, 4
}
```

`continue` skips the rest of the current iteration and jumps back to the condition check:

```
var number = 0

while number < 6 {
    number += 1
    if number % 2 == 0 {
        continue
    }
    print(number) // 1, 3, 5
}
```

Here every even number gets skipped, because `continue` jumps over the `print()` call.

The infinite loop pitfall

The most common `while` bug is forgetting to change the variable used in the condition:

```
var count = 0

while count < 3 {
    print(count)
    // we forgot count += 1
}
```

`count` stays at 0 forever, so the condition never becomes false. This loop prints 0 until you kill the program.

My advice: every time you write a `while` loop, ask yourself "what makes this condition become false?" before you write the body. If you can't answer, the loop will never end.

Check your understanding: control flow

Check the key ideas from control flow before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Collections and functions

Organize values with arrays, sets, dictionaries, and reusable functions.

Arrays in Swift

An introduction to arrays in Swift: how to declare and type them, access items, and use methods like append, insert, remove, count, isEmpty, and sort.

This tutorial belongs to the Swift series

We use arrays to create a collection of items.

In this example we create an array holding 3 integers:

```
var list = [1, 2, 3]
```

We can access the first item using the syntax `list[0]`, the second using `list[1]` and so on.

Elements in an array in Swift must have the same type.

The type can be inferred if you initialize the array at declaration time, like in the case above.

Otherwise the type of values an array can include must be declared, in this way:

```
var list: [Int] = []
```

Another shorthand syntax is:

```
var list = [Int]()
```

You can also explicit the type at initialization, like this:

```
var list: [Int] = [1, 2, 3]
```

A quick way to initialize an array is to use the range operator:

```
var list = Array(1...4) //[1, 2, 3, 4]
```

To get the number of items in the array, use the `count` property:

```
var list = [1, 2, 3]
list.count //3
```

If an array is empty, its `isEmpty` property is `true`.

```
var list = [1, 2, 3]
list.isEmpty //false
```

You can append an item at the end of the array using the `append()` method:

```
var list: [Int] = [1, 2, 3]
list.append(4)
```

or you can an item at any position of the array using `insert(newElement: <Type> at: Int)`:

```
var list: [Int] = [1, 2, 3]
list.insert(17, at: 2)
//list is [1, 2, 17, 3]
```

> An array must be declared as `var` to be modified. If it's declared with `let`, you cannot modify it by adding or removing elements.

To remove one item from the array, use `remove(at:)` passing the index of the element to remove:

```
var list: [Int] = [1, 2, 3]
list.remove(1)
//list is [1, 3]
```

`removeLast()` and `removeFirst()` are two handy ways to remove the last and first element.

To remove all items from the array, you can use removeAll() or you can assign an empty array:

```
var list: [Int] = [1, 2, 3]
list.removeAll()
//or
list = []
```

The `sort()` method sorts the array:

```
var list = [3, 1, 2]
list.sort()
//list is [1, 2, 3]
```

There are a lot more methods, but those are the basic ones.

Arrays are equal when they contain the same elements, of the same type:

```
[1, 2, 3] == [1, 2, 3] //true
```

Arrays are passed by value, which means if you pass an array to a function, or return it from a function, the array is copied.

Arrays are collections, and they can be iterated over in loops.

Swift Sets

Learn how to use sets in Swift to store unique, unordered items, from insert and contains to count, plus set math like union and intersection.

This tutorial belongs to the Swift series

Sets are used to create collections of non-repeated items.

While an array can contain many times the same item, you only have unique items in a set.

You can declare a set of Int values in this way:

```
let set: Set<Int> = [1, 2, 3]
```

or you can initialize it from an array:

```
let set = Set([1, 2, 3])
```

Add items to the set using `insert()`:

```
var set = Set([1, 2, 3])
set.insert(17)
```

Unlike arrays, there is no order or position in a set. Items are retrieved and inserted randomly.

The way to print the content of a set ordered is to transform it into an array using the `sorted()` method:

```
var set = Set([2, 1, 3])
let orderedList = set.sorted()
```

To check if a set contains an element, use the `contains()` method:

```
var set = Set([1, 2, 3])
set.contains(2) //true
```

To get the number of items in the set, use the `count` property:

```
let set = Set([1, 2, 3])
set.count //3
```

If a set is empty, its `isEmpty` property is `true`.

```
let set = Set([1, 2, 3])
set.isEmpty //false
```

To remove one item from the array, use `remove()` passing the value of the element:

```
var set = Set([1, 2, 3])
set.remove(1)
//set is [2, 3]
```

To remove all items from the set, you can use removeAll():

```
set.removeAll()
```

Sets, like arrays, are passed by value, which means if you pass it to a function, or return it from a function, the set is copied.

Sets are great to perform set math operations like intersection, union, subtracting, and more.

These methods help with this:

- `intersection(_:)`
- `symmetricDifference(_:)`
- `union(_:)`
- `subtracting(_:)`
- `isSubset(of:)`
- `isSuperset(of:)`
- `isStrictSubset(of:)`
- `isStrictSuperset(of:)`
- `isDisjoint(with:)`

Sets are collections, and they can be iterated over in loops.

Swift Dictionaries

An introduction to dictionaries in Swift: how to create key-value pairs, read and change values by key, add and remove entries, and use count and isEmpty.

> This tutorial belongs to the Swift series

We use dictionaries to create a collection of key-value pairs.

Here is how to create a dictionary with 1 key-value pairs, where the key is a String and the value is an Int:

```
var dict = ["Roger": 8, "Syd": 7]
```

In this case the type is inferred. You can also explicitly set the type at declaration time:

```
var dict: [String: Int] = ["Roger": 8, "Syd": 7]
```

In this example we create an empty dictionary of Int keys and String values:

```
var dict = [String: Int]()

//or

var dict: [String: Int] = [:]
```

You can access the value assigned to a key using this syntax:

```
var dict = ["Roger": 8, "Syd": 7]

dict["Roger"] //8
dict["Syd"] //7
```

You can change the value assigned to a key in this way:

```
dict["Roger"] = 9
```

> A dictionary must be declared as `var` to be modified. If it's declared with `let`, you cannot modify it by adding or removing elements.

Use the same syntax to add a new key/value pair:

```
dict["Tina"] = 4
```

To remove a key/value paid, assign the value to `nil`:

```
dict["Tina"] = nil
```

Or call the `removeValue(forKey:)` method:

```
dict.removeValue(forKey: "Tina")
```

To get the number of items in the dictionary, use the `count` property:

```
var dict = ["Roger": 8, "Syd": 7]
dict.count //2
```

If a dictionary is empty, its `isEmpty` property is `true`.

```
var dict = [String: Int]()
dict.isEmpty //true
```

There are a lot of methods related to dictionaries, but those are the basic ones.

Dictionaries are passed by value, which means if you pass it to a function, or return it from a function, the dictionary is copied.

Dictionaries are collections, and they can be iterated over in loops.

Swift Functions

Learn how to declare functions in Swift with the func keyword, pass labeled parameters, return values, and use tuples to return more than one value.

This tutorial belongs to the Swift series

Your program's code is organized into functions.

A function is declared using the `func` keyword:

```
func bark() {
    print("woof!")
}
```

Functions can be assigned to structures, classes and enumerations, and in this case we call them methods.

A function is invoked using its name:

```
bark()
```

A function can return a value:

```
func bark() -> String {
    print("woof!")
      return "barked successfully"
}
```

And you can assign it to a variable:

```
let result = bark()
```

A function can accept parameters. Each parameter has a name, and a type:

```
func bark(times: Int) {
```

```
    for index in 0..<times {
        print("woof!")
    }
}
```

The name of a parameter is internal to the function.

We use the name of the parameter when we call the function, to pass in its value:

```
bark(times: 3)
```

When we call the function we must pass all the parameters defined.

Here is a function that accepts multiple parameters:

```
func bark(times: Int, repeatBark: Bool) {
    for index in 0..<times {
        if repeatBark == true {
            print("woof woof!")
        } else {
            print("woof!")
        }
    }
}
```

In this case you call it in this way:

```
bark(times: 3, repeat: true)
```

When we talk about this function, we don't call it `bark()`. We call it `bark(times:repeat:)`.

This is because we can have multiple functions with the same name, but different set of parameters.

You can avoid using labels by using the `_` keyword:

```
func bark(_ times: Int, repeatBark: Bool) {
    //...the function body
}
```

So you can invoke it in this way:

```
bark(3, repeat: true)
```

It's common in Swift and iOS APIs to have the first parameter with no label, and the other parameters labeled.

It makes for a nice and expressive API, when you design the names of the function and the parameters nicely.

You can only return one value from a function. If you need to return multiple values, it's common to return a tuple:

```
func bark() -> (String, Int) {
    print("woof!")
      return ("barked successfully", 1)
}
```

And you can assign the result to a tuple:

```
let (result, num) = bark()

print(result) //"barked successfully"
print(num) //1
```

Functions can be nested inside other functions. When this happens, the inner function is invisible to outside the outer function.

Check your understanding: collections and functions

Check the key ideas from collections and functions before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Model your program

Choose structures, enumerations, classes, and protocols for clear program models.

Swift Structures

Learn how structures work in Swift, from defining stored properties and methods to why structs are value types that get copied when passed around.

This tutorial belongs to the Swift series

Structures are one essential Swift concepts.

Structures are everywhere in Swift. Even the built-in types are structures.

We can create instances of structures, which we call **objects**.

In most languages, objects can only be created from classes. Swift has classes, too, but you can create objects also from structures and the official documentation advises to prefer structures because they are easier to use.

They are a light versions of classes.

A struct can have properties. A struct can have methods (functions) A struct can define subscripts A struct can define initializers A struct can conform to protocols A struct can be extended

One important thing classes allow is inheritance, so if you need that, you have classes.

A struct is defined using this syntax:

```
struct Dog {

}
```

Inside a structure you can define **stored properties**:

```
struct Dog {
    var age = 8
    var name = "Roger"
}
```

This structure definition defines a **type**. To create a new instance with this type, we use this syntax:

```
let roger = Dog()
```

Once you have an instance, you can access its properties using the dot syntax:

```
let roger = Dog()
roger.age
roger.name
```

The same dot syntax is used to update a property value:

```
roger.age = 9
```

You can also create a struct instance passing the values of the properties:

```
let syd = Dog(age: 7, name: "Syd")
syd.age
syd.name
```

To do so, properties must be defined variables, with `var`, not as constants (with `let`). It's also important to respect the order those properties are defined.

Structures can have **instance methods**: functions that belong to an instance of a structure.

```
struct Dog {
    var age = 8
    var name = "Roger"
    func bark() {
        print("\(name): wof!")
    }
}
```

And we also have **type methods**:

```
struct Dog {
    var age = 8
    var name = "Roger"
    func bark() {
        print("\(name): wof!")
    }
    static func hello() {
        print("Hello I am the Dog struct")
    }
}
```

Invoked as `Dog.hello()`

Structures are a value type. This means they are copied when passed to a function, or when returned from a function. And when we assign a variable pointing to a structure to another variable.

This also means that if we want to update the properties of a structure we must define it using `var` and not `let`.

All types in Swift are defined as structures: Int, Double, String, arrays and dictionaries, and more, are structures.

Swift Enumerations

An introduction to enumerations in Swift: how to group options under a type, use them in switch statements, and assign raw values you read with rawValue.

This tutorial belongs to the Swift series

Enumerations are a way to group a set of different options, under a common name.

Example:

```
enum Animal {
    case dog
    case cat
    case mouse
    case horse
}
```

This `Animal` enum is now a **type**.

A type whose value can only be one of the cases listed.

If you define a variable of type `Animal`:

```
var animal: Animal
```

you can later decide which value to assign it using this syntax:

```
var animal: Animal
animal = .dog
```

We can use enumerations in control structures like switches:

```
enum Animal {
    case dog
    case cat
    case mouse
    case horse
}

let animal = Animal.dog

switch animal {
case .dog: print("dog")
case .cat: print("cat")
default: print("another animal")
}
```

Enumerations values can be strings, characters or numbers.

You can also define an enum on a single line:

```
enum Animal {
    case dog, cat, mouse, horse
}
```

And you can also add type declaration to the enumeration, and each case has a value of that type assigned:

```
enum Animal: Int {
    case dog = 1
    case cat = 2
    case mouse = 3
    case horse = 4
}
```

Once you have a variable, you can get this value using its `rawValue` property:

```
enum Animal: Int {
    case dog = 1
    case cat = 2
    case mouse = 3
    case horse = 4
}

var animal: Animal
animal = .dog

animal.rawValue //1
```

Enumerations are a value type. This means they are copied when passed to a function, or when returned from a function. And when we assign a variable pointing to an enumeration to another variable.

Swift Classes

An introduction to classes in Swift: defining properties and methods, why classes are reference types, writing initializers with self, and inheritance.

This tutorial belongs to the Swift series

Classes are a bit similar to structures, but they have some key differences.

A class is defined using this syntax:

```
class Dog {

}
```

Inside a class you can define stored properties:

```
class Dog {
    var age = 0
}
```

A class definition defines a **type**. To create a new instance with this type, we use this syntax:

```
let roger = Dog()
```

Once you have an instance, you can access its properties using the dot syntax:

```
let roger = Dog()
roger.age
```

The same dot syntax is used to update a property value:

```
roger.age = 9
```

One big difference is that classes are reference types. Structures (and enumerations) are value types.

This means that assigning a class instance to another variable does not copy the instance. Both variables point to the same instance:

```
class Dog {
    var age = 0
}

let roger = Dog()
let syd = roger

roger.age = 9
//syd.age == 9
```

This also means we can define a reference to a class using `let`, and we can change its properties, as you saw in the example above.

We can create instances of classes, and we call them **objects**.

As with structs, classes can have properties, methods, and more.

Contrary to structs, we **must** define an initializer in order to initialize the values when we create an instance:

```
class Dog {
    var age : Int

    init(age: Int) {
        self.age = age
    }
}

let roger = Dog(age: 9)
```

You can only declare properties without initializing them if you have an initializer.

See the use of `self`. We need it because `age` is both an instance property and the `init(age:)` method parameter. `self.age` references the `age` instance property.

Classes can have **instance methods**: functions that belong to an instance of a class.

```
class Dog {
    var age = 8
    var name = "Roger"

    func bark() {
      print("\(name): wof!")
    }
}
```

And we also have **type methods**:

```
class Dog {
```

```
    var age = 8
    var name = "Roger"

    func bark() {
        print("\(name): wof!")
    }
    static func hello() {
        print("Hello I am the Dog struct")
    }
}
```

Invoked as `Dog.hello()`

One important thing classes allow is inheritance.

A class can inherit all the properties and methods from another class.

Say we have a class `Animal`. Every animal has an age:

```
class Animal {
    var age: Int
}
```

Not every animal has a name. Dogs have a name. So we create a `Dog` class extending from `Animal`:

```
class Dog: Animal {
    var name: String
}
```

Now we must add an initializer for both classes. In the Dog case, after we do the class-specific initialization, we can call the parent class initializer using `super.init()`:

```
class Animal {
    var age: Int

    init(age: Int) {
        self.age = age
    }
}

class Dog: Animal {
    var name: String

    init(age: Int, name: String) {
        self.name = name
        super.init(age: age)
    }
}

var horse = Animal(age: 8)
var roger = Dog(age: 8, name: "Roger")
```

`Animal` is now a **superclass**, and `Dog` is a **subclass**.

There's more to say about classes, but this is a good introduction.

Swift Protocols

Learn how protocols work in Swift to give different types a shared set of properties and methods, and how a struct or class adopts and conforms to them.

This tutorial belongs to the Swift series

A protocol is a way to have different objects, of different types, have a common set of functionality.

A protocol is defined in this way:

```
protocol Mammal {

}
```

Structs and classes can **adopt a protocol** in this way:

```
struct Dog: Mammal {

}

class Cat: Mammal {

}
```

A protocol can define properties and methods, without providing values and implementations, and a struct/class must implement them:

```
protocol Mammal {
    var age: Int { get set }
    func walk()
}
```

The property can be defined as `get` or `get set`. If it's `get`, the property must be implemented as read only, with a getter.

Any type that adopts the protocol must **conform** to the protocol by implementing those methods or providing those properties:

```
struct Dog: Mammal {
    var age: Int = 0
    func walk() {
        print("The dog is walking")
    }
}

class Cat: Mammal {
    var age: Int = 0
    func walk() {
        print("The cat is walking")
    }
}
```

Structs and classes can adopt multiple protocols:

```
struct Dog: Mammal, Animal {

}

class Cat: Mammal, Animal {

}
```

Notice that for classes, this is the same syntax used to define a superclass. If there is a superclass, list it as the first item in the list, after the colon.

Check your understanding: modeling a program

Check the key ideas from modeling a program before continuing.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/swift/.