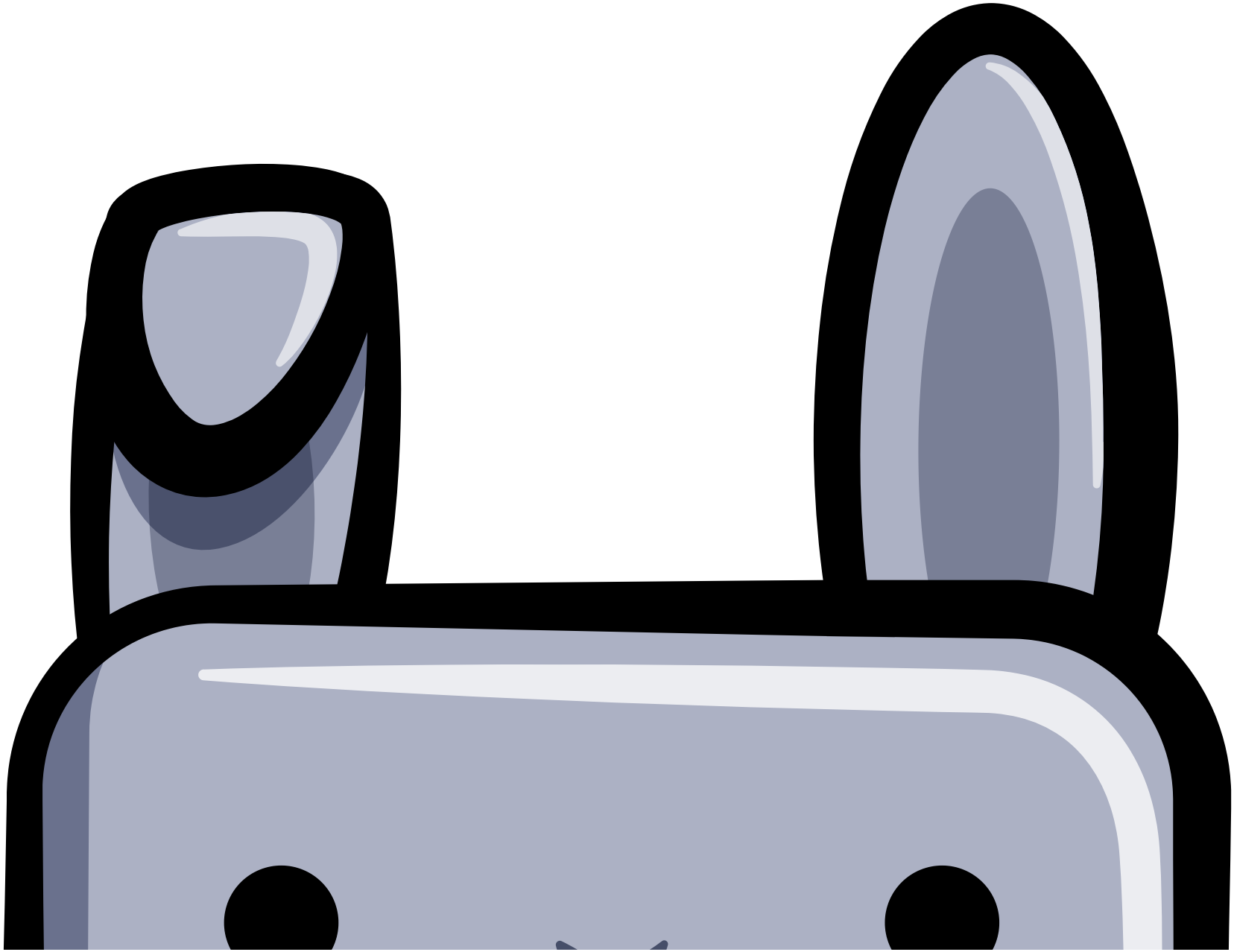
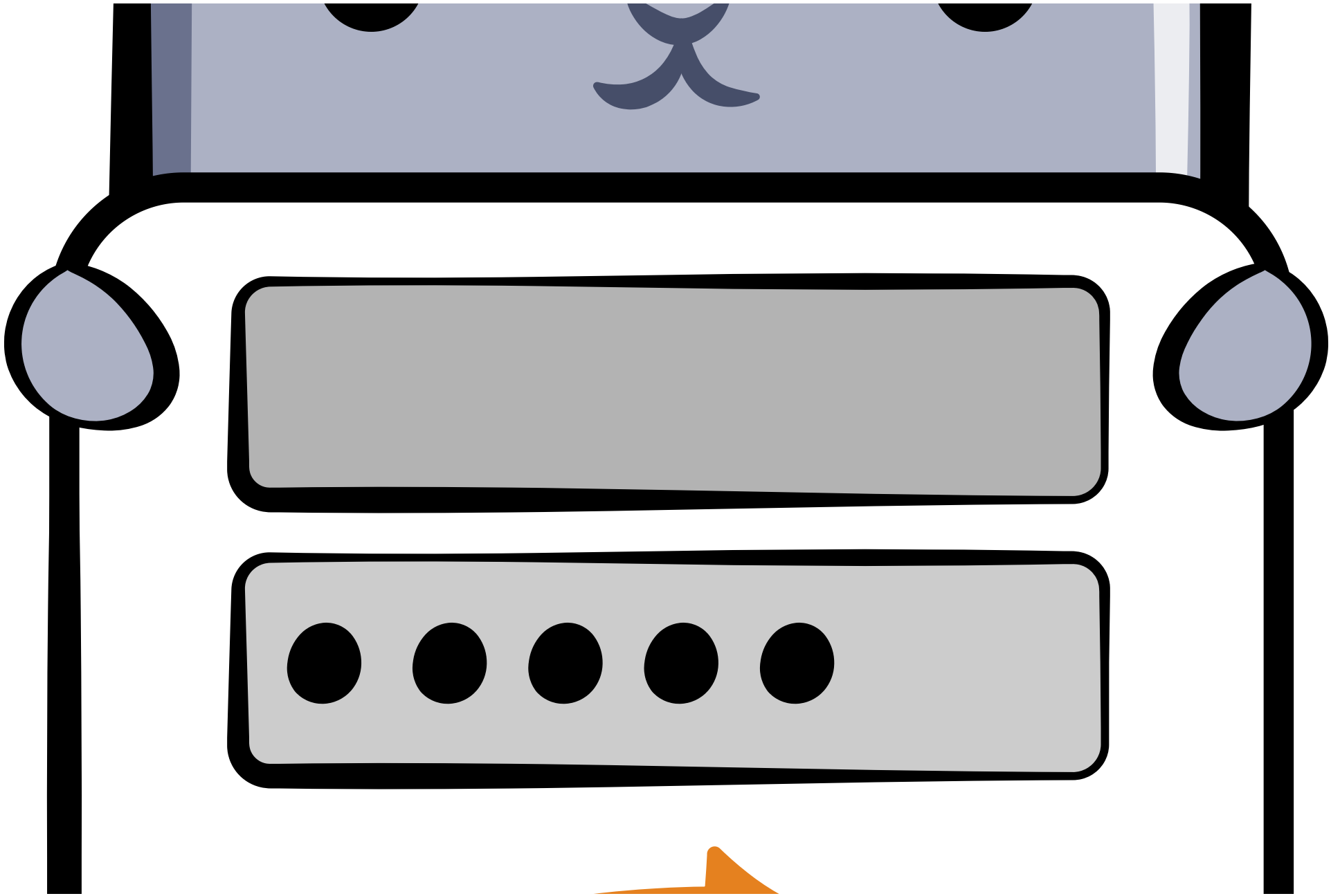


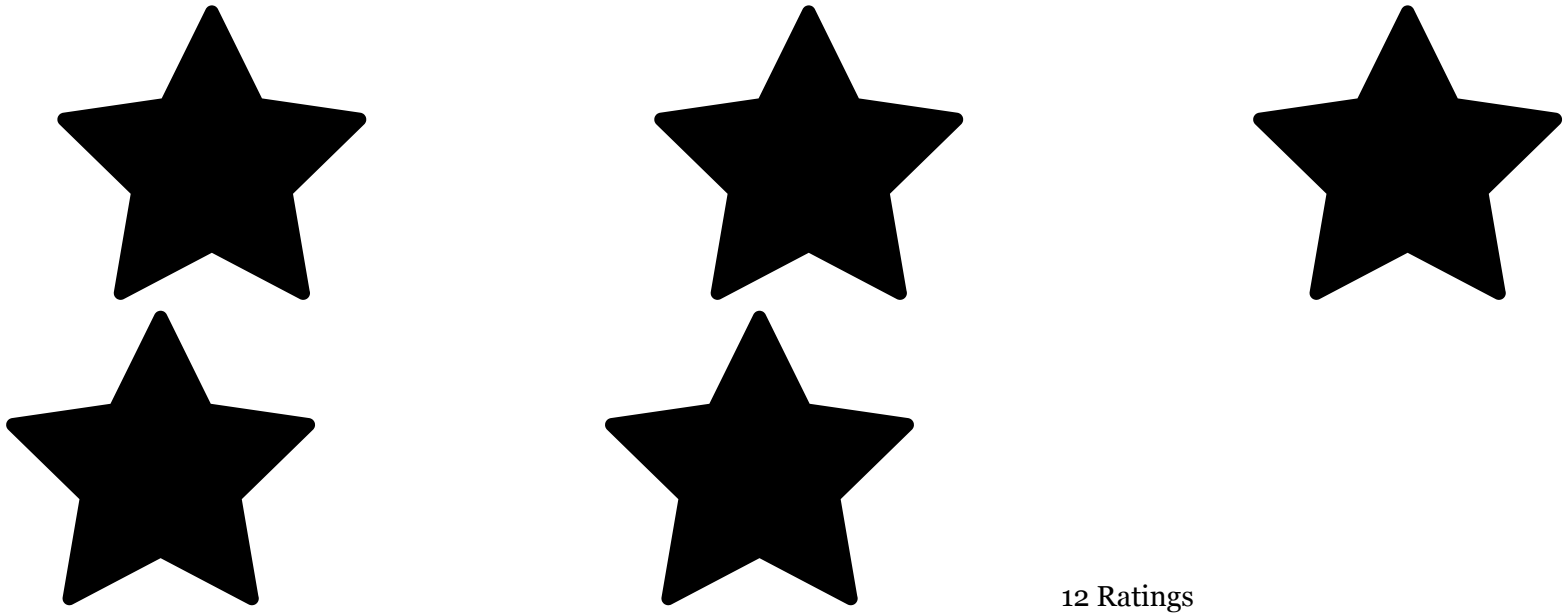


Get Started
Home · iOS & Swift Tutorials

Swift Tutorial Part 3: Flow Control

Welcome to part 3 of our Swift tutorial, where you'll learn how code decisions using Booleans and repeat tasks using loops to control the flow.

By Lorenzo Boaro Oct 5 2018 · Article (35 mins) · Beginner

5/5 

12 Ratings

Update note: Lorenzo Boaro updated this tutorial to iOS 12, Xcode 10, and Swift 4.2. Matt Galloway wrote the original.

Welcome to the final part of this Swift mini-series! If you haven't read through Part 1 or Part 2 of this series, we suggest you do that first.

In computer programming terms, the ability to tell the computer what to do in different scenarios is known as *control flow*.

In this tutorial, you'll learn how to make decisions and repeat tasks in your programs by using syntax to control the flow. You'll also learn about *Booleans*, which represent `true` and `false` values, and how you can use these to compare data.

Getting Started

At this point, you've seen a few types, such as `Int`, `Double` and `String`. Here, you'll learn about another type — one that will let you compare values through the *comparison operators*.

When you perform a comparison, such as looking for the greater of two numbers, the answer is either *true* or *false*. Swift has a data type just for this! It's called `Bool`, which is short for Boolean, after George Boole who invented an entire field of mathematics around the concept of `true` and `false`.

This is how you use a Boolean in Swift:

```
let yes = true // Inferred to be of type Bool
let no = false // Inferred to be of type Bool
```

A Boolean can only be either `true` or `false`, denoted by the keywords `true` and `false`. In the code above, you use the keywords to set the value of each constant.

Boolean Operators

Booleans are commonly used to compare values. For example, you may have two values and you want to know if they're equal: Either the values are the same (true) or they are different (false).
In Swift, you do this using the *equality operator*, which is denoted by ==:

```
let doesOneEqualTwo = (1 == 2)
```

Swift infers that `doesOneEqualTwo` is a `Bool`. Clearly, 1 does not equal 2, and therefore `doesOneEqualTwo` will be `false`.
Similarly, you can find out if two values are *not* equal using the `!=` operator:

```
let doesOneNotEqualTwo = (1 != 2)
```

This time, the comparison is `true` because 1 does not equal 2, so `doesOneNotEqualTwo` will be `true`.
The prefix `!` operator, also called the *not operator*, toggles `true` to `false` and `false` to `true`. Another way to write the above is:

```
let alsoTrue = !(1 == 2)
```

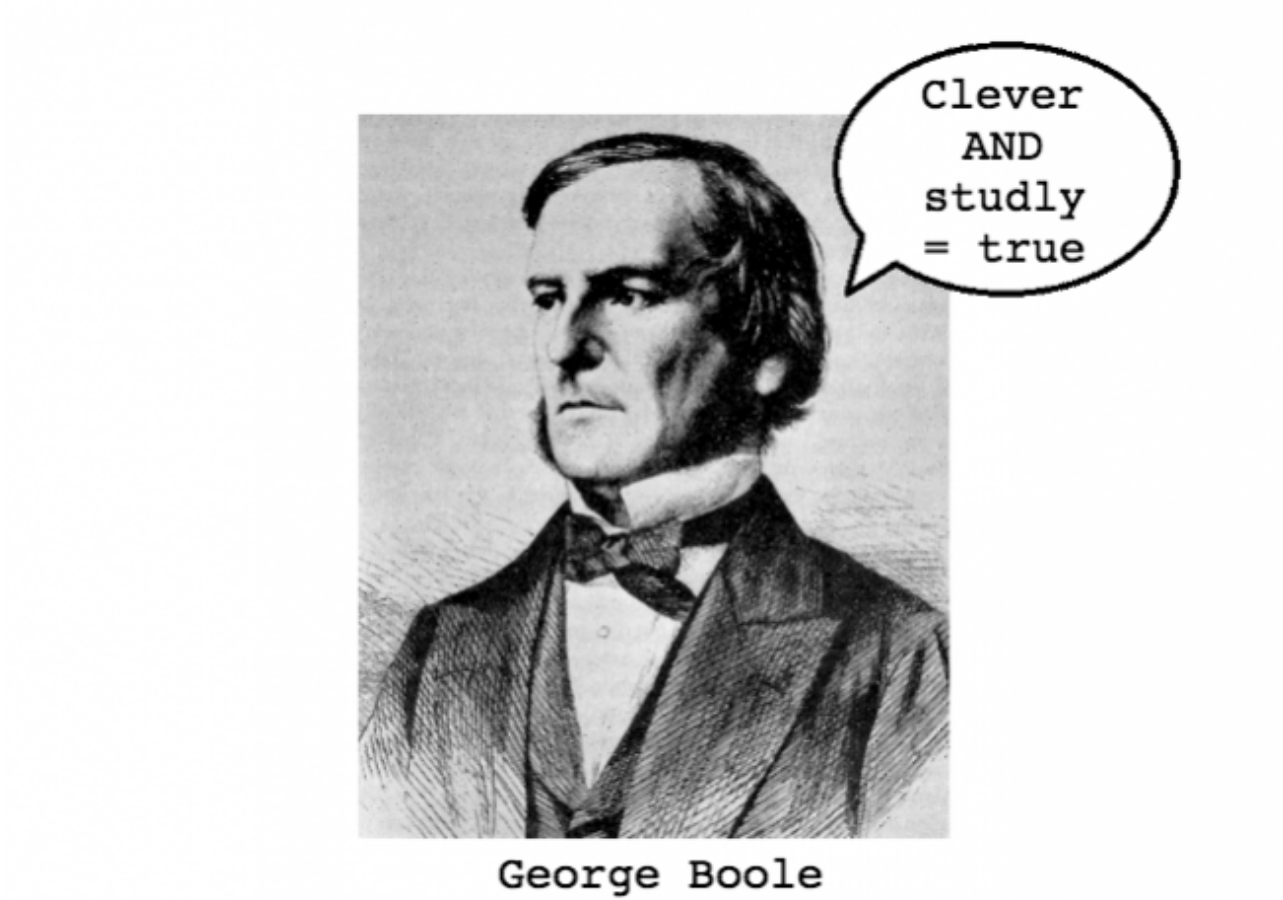
Evaluating this step by step: 1 does not equal 2, so (1==2) is `false`, and then `!` toggles `false` to `true`.
Two more operators let you determine if a value is greater than (>) or less than (<) another value. You'll likely know these from mathematics:

```
let isOneGreaterThanTwo = (1 > 2)
let isOneLessThanTwo = (1 < 2)
```

It's not rocket science to work out that `isOneGreaterThanTwo` will equal `false` and `isOneLessThanTwo` will equal `true`.
There's also an operator that lets you test if a value is less than *or* equal to another value: <=. It's a combination of < and ==, and will therefore return `true` if the first value is either less than the second value or equal to it.
Similarly, >= operator lets you test if a value is greater than or equal to another.

Boolean Logic

The examples above test just one condition. In order to combine multiple conditions to form a result, you can rely on Boolean logic.
One way to combine conditions is by using *AND*. When you AND together two Booleans, the result is another Boolean. If both input Booleans are `true`, then the result is `true`. Otherwise, the result is `false`.



In Swift, the operator for Boolean AND is `&&`, used like so:

```
let and = true && true
```

In this case, `and` will be `true`. If either of the values on the right was `false`, then `and` would be `false`.
Another way to combine conditions is by using *OR*. When you OR together two Booleans, the result is `true` if *either* of the input Booleans is `true`. Only if *both* input Booleans are `false` will the result be `false`.
In Swift, the operator for Boolean OR is `||`, used like so:

```
let or = true || false
```

In this case, `or` will be `true`. If both values on the right were `false`, then `or` would be `false`. If both were `true`, then `or` would still be `true`.
In Swift, Boolean logic is usually applied to multiple conditions. For example, consider the following:

```
let andTrue = 1 < 2 && 4 > 3
let andFalse = 1 < 2 && 3 > 4

let orTrue = 1 < 2 || 3 > 4
let orFalse = 1 == 2 || 3 == 4
```

Each of these tests two separate conditions, combining them with either AND or OR.
It's also possible to use Boolean logic to combine more than two comparisons. For example, you can form a complex comparison like so:

```
let andOr = (1 < 2 && 3 > 4) || 1 < 4
```

When you include parentheses around part of the expression, you specify the order of evaluation. First, Swift evaluates the sub-expression inside the parentheses, and then it evaluates the full expression, as follows:

```
1. (1 < 2 && 3 > 4) || 1 < 4
2. (true && false) || true
3. false || true
4. true
```

String Equality

Sometimes, you want to determine if two strings are equal. For example, a children's game of naming an animal in a photo would need to determine if the player answered correctly.
In Swift, you can compare strings using the standard equality operator, ==, in exactly the same way as you compare numbers. For example:

```
let guess = "dog"
let dogEqualsCat = guess == "cat"
```

`dogEqualsCat` is a Boolean that in this case equals `false`, because `"dog"` does not equal `"cat"`.
Just as with numbers, you can compare not just for equality, but also to determine is one value is greater than or less that another value. For example:

```
let order = "cat" < "dog"
```

This syntax checks if one string comes before another alphabetically. In this case, `order` equals `true` because `"cat"` comes before `"dog"`.

The if Statement

The first and most common way of controlling the flow of a program is through the use of an *`if` statement*, which allows the program to do something only *if* a certain condition evaluates as `true`. For example, consider the following:

```
if 2 > 1 {
  print("Yes, 2 is greater than 1.")
}
```

This is a simple `if` statement. If the condition is `true`, then the statement will execute the code between the braces. If the condition evaluates to `false`, then the statement won't execute the code between the braces.
You can extend an `if` statement to provide code to run in case the condition turns out to be `false`. This is known as the *else clause*. For example:

```
let animal = "Fox"
if animal == "Cat" || animal == "Dog" {
  print("Animal is a house pet.")
} else {
  print("Animal is not a house pet.")
}
```

Here, if `animal` equals either `"Cat"` or `"Dog"`, then the statement will run the first block of code. If `animal` does not equal either `"Cat"` or `"Dog"`, then the statement will run the block inside the `else` part of the `if` statement, printing the following to the debug area:
Animal is not a house pet.
But you can go even further than that with `if` statements. Sometimes, you want to check one condition, then another. This is where *`else-if`* comes into play, nesting another `if` statement in the `else` clause of a previous `if` statement.
You can use it like so:

```
let hourOfDay = 12
var timeOfDay = ""

if hourOfDay < 6 {
  timeOfDay = "Early morning"
} else if hourOfDay < 12 {
  timeOfDay = "Morning"
} else if hourOfDay < 17 {
  timeOfDay = "Afternoon"
} else if hourOfDay < 20 {
  timeOfDay = "Evening"
} else if hourOfDay < 24 {
  timeOfDay = "Late evening"
} else {
  timeOfDay = "INVALID HOUR!"
}
print(timeOfDay)
```

These nested `if` statements test multiple conditions one-by-one until a `true` condition is found. Only the code associated with that first `true` condition is executed, regardless of whether subsequent `else-if` conditions are `true`. In other words, the order of your conditions matters!

You can add an `else` clause at the end to handle the case in which none of the conditions are `true`. This `else` clause is optional if you don't need it; in this example, you *do* need it to ensure that `timeOfDay` has a valid value by the time you print it out.

In this example, the `if` statement takes a number representing an hour of the day and converts it to a string that represents the part of the day to which the hour belongs. Working with a `24`-hour clock, the statements are checked one-by-one in order.

In the code above, the `hourOfDay` constant is `12`. Therefore, the code will print the following:

Afternoon

Notice that, even though both the `hourOfDay < 20` and `hourOfDay < 24` conditions are also `true`, the statement only executes the first block whose condition is `true`; in this case, the block with the `hourOfDay < 17` condition.

Encapsulating Variables

`if` statements introduce a new concept *scope*, which is a way to encapsulate variables through the use of braces.

Imagine that you want to calculate the fee to charge your client. Here's the deal you've made:

You earn $25 for every hour up to 40 hours, and $50 for every hour thereafter.
Using Swift, you can calculate your fee in this way:

```
var hoursWorked = 45

var price = 0
if hoursWorked > 40 {
  let hoursOver40 = hoursWorked - 40
  price += hoursOver40 * 50
  hoursWorked -= hoursOver40
}
price += hoursWorked * 25

print(price)
```

This code takes the number of hours and checks if it's over 40. If so, the code calculates the number of hours over 40 and multiplies that by $50, then adds the result to the price. The code then subtracts the number of hours over 40 from the hours worked. It multiplies the remaining hours worked by $25 and adds that to the total price.
In the example above, the result is as follows:
1250
`hoursOver40` declares a new constant that you can use inside the `if` statement. But what happens if you try to use it at the end of the above code?

```
// ...

print(price)
print(hoursOver40)
```

This would result in the following error:
`Use of unresolved identifier 'hoursOver40'`
This error informs you that you're only allowed to use the `hoursOver40` constant within the scope in which it was created. In this case, the `if` statement introduced a new scope, so when that scope is finished, you can no longer use the constant.
However, each scope can use variables and constants from its parent scope. In the example above, the scope inside of the `if` statement uses the `price` and `hoursWorked` variables, which you created in the parent scope.

The Ternary Conditional Operator

The ternary conditional operator takes a condition and returns one of two values, depending on whether the condition was `true` or `false`.
The syntax is as follows:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

Here's an example that tests the minimum of two variables:

```
let a = 5
let b = 10

let min: Int
if a < b {
  min = a
} else {
  min = b
}
```

Thanks to the ternary operator, you can rewrite your code above like so:

```
let a = 5
let b = 10

let min = a < b ? a : b
```

If the condition `a < b` is `true`, then the result assigned back to `min` will be the value of `a`; if it's `false`, the result will be the value of `b`.
Note: Use ternary operator with care. Its conciseness can lead to hard-to-read code if overused.

Loops

Loops are a way of executing code multiple times. In this section, you'll learn about the `while` loop.

While Loops

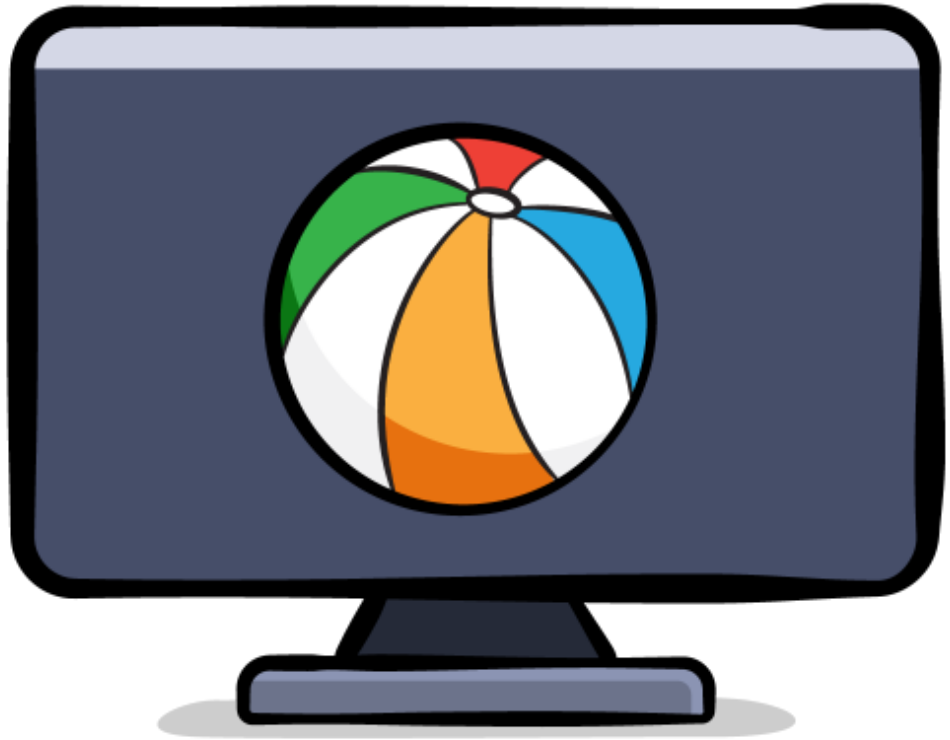
A `while` loop repeats a block of code while a condition is `true`.
You create a `while` loop this way:

```
while <CONDITION> {
  <LOOP CODE>
}
```

During every iteration, the loop checks the condition. If the condition is `true`, then the loop executes and moves on to another iteration. If the condition is `false`, then the loop stops. Just like `if` statements, `while` loops introduce a scope.
The simplest `while` loop takes this form:

```
while true {
}
```

This is a `while` loop that never ends because the condition is always `true`. An *infinite loop* might not cause your program to crash, but it will very likely cause your computer to freeze.



Here's a more useful example of a `while` loop:

```
var sum = 1

while sum < 1000 {
  sum = sum + (sum + 1)
}
```

This code calculates a mathematical sequence, up to the point in which the value is greater than `1000`.
The loop executes as follows:

After the ninth iteration, the `sum` variable is `1023` and, therefore, the loop condition of `sum < 1000` becomes `false`. At this point, the loop stops.

Repeat-while Loops

A variant of the `while` loop is called the *repeat-while loop*. It differs from the `while` loop in that the condition is evaluated *at the end* of the loop rather than at the beginning.
You construct a `repeat-while` loop like this:

```
repeat {
  <LOOP CODE>
} while <CONDITION>
```

Here's the example from the last section, but using a `repeat-while` loop:

```
sum = 1

repeat {
  sum = sum + (sum + 1)
} while sum < 1000
```

In this example, the outcome is the same as before. However, that isn't always the case — you might get a different result with a different condition.
Consider the following `while` loop:

```
sum = 1

while sum < 1 {
  sum = sum + (sum + 1)
}
```

And now consider the corresponding `repeat-while` loop, which uses the same condition:

```
sum = 1

repeat {
  sum = sum + (sum + 1)
} while sum < 1
```

In the case of the regular `while` loop, the condition `sum < 1` is `false` right from the start. That means the body of the loop won't be reached! The value of `sum` will equal `1` because the loop won't execute any iterations.
In the case of the `repeat-while` loop, however, `sum` will equal `3` because the loop will execute once.

Breaking Out of a Loop

Sometimes you want to break out of a loop early. You can do this using the `break` statement, which immediately stops the execution of the loop and continues on to the code after the loop.
For example, consider the following code:

```
sum = 1

while true {
  sum = sum + (sum + 1)
  if sum >= 1000 {
    break
  }
}
```

Here, the loop condition is `true`, so the loop would normally iterate forever. However, the `break` means the `while` loop will exit once the sum is greater than or equal to `1000`.

Advanced Control Flow

In this section, you'll continue to learn how to control the flow of execution. You'll learn about another loop known as the `for` loop.

Ranges

Before you dive into the `for` loop statement, you need to know about the `ClosedRange` and `Range` types, which lets you represent a sequence of numbers.
First, there's *closed range*, which you represent like so:

```
let closedRange = 0...5
```

The three dots (`...`) indicate that this range is closed, which means the range goes from 0 to 5, inclusive. That means the numbers `(0, 1, 2, 3, 4, 5)`.
Second, there's half-open range, which you represent like so:

```
let halfOpenRange = 0..<5
```

Here, you replace the three dots with two dots and a less-than sign (`..<`). Half open means the range goes from 0 to 5, inclusive of 0 but *not* of 5. That means the numbers `(0, 1, 2, 3, 4)`.
Both closed and half-open ranges must always be increasing. In other words, the second number must always be greater than or equal to the first.
Ranges are commonly used in both `for` loops and `switch` statements, which means you'll use ranges soon!

For Loops

Now that you know about ranges, it's time to look at the `for` loop. It's used to run code a certain number of times.
You construct a `for` loop like this:

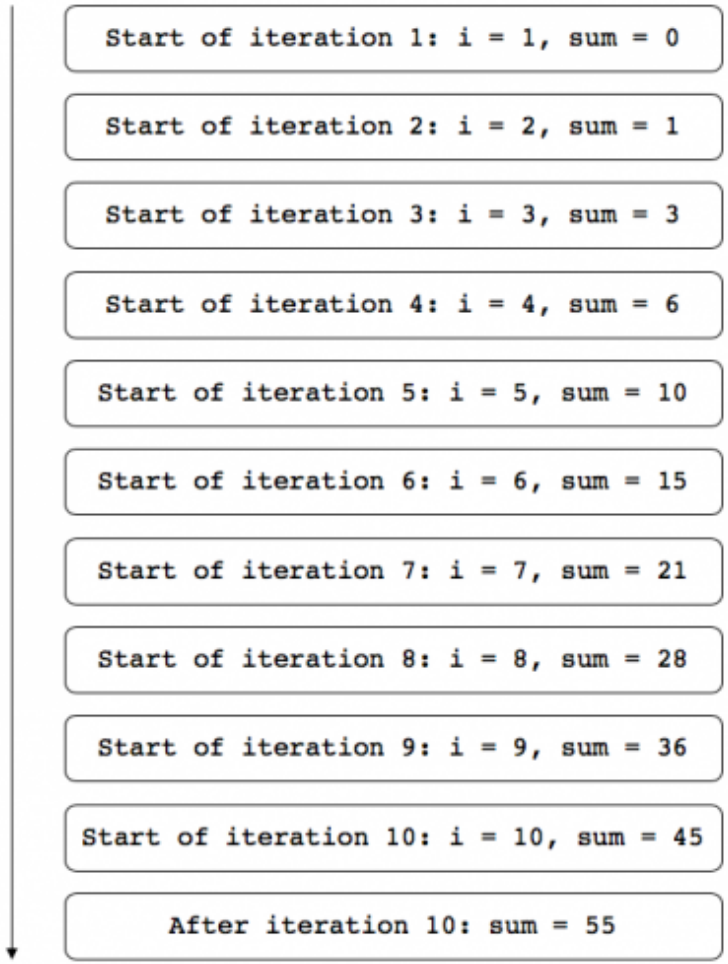
```
for <CONSTANT> in <RANGE> {
  <LOOP CODE>
}
```

The loop begins with the `for` keyword, followed by a name given to the loop constant, followed by `in`, followed by the range to loop through. Here's an example:

```
let count = 10
var sum = 0

for i in 1...count {
  sum += i
}
```

The `for` loop iterates through the range 1 to `count`. At the first iteration of the loop, `i` will equal the first element in the range — 1. Each time around the loop, `i` will increment until it's equal to `count`; the loop will execute one final time and then finish.
Note: If you'd used a half-open range, then the last iteration would see `i` equal to `count - 1`.
Inside the loop, you add `i` to the `sum` variable; it runs 10 times to calculate the sequence `1 + 2 + 3 + 4 + 5 + ...` all the way up to 10.
Here are the values of the constant `i` and variable `sum` for each iteration:



The `i` constant is only visible inside the scope of the `for` loop, which means it's not available outside of the loop.
When you are not interested in the loop constant at all, you can employ the underscore to indicate that you're ignoring it. For example:

```
sum = 1
var lastSum = 0

for _ in 0..<count {
  let temp = sum
```

```
  sum = sum + lastSum
  lastSum = temp
}
```

This code doesn't require a loop constant; the loop simply needs to run a certain number of times. In this case, the range is `0` through `count` and is half open. This is the usual way of writing loops that run a certain number of times.
It's also possible to only perform the iteration under certain conditions. For example, imagine that you wanted to compute a sum but only for odd numbers:

```
var sum = 0
for i in 1...count where i % 2 == 1 {
  sum += i
}
```

The loop above has a `where` clause in the `for` loop statement. The loop still runs through all values in the range `1` to `count`, but it will only execute the loop's code block when the `where` condition is `true`; in this case, it executes when `i` is odd.

Continue Statement

Sometimes you'd like to skip a loop iteration for a particular case without breaking out of the loop entirely. You can do this with the `continue` statement, which immediately ends the current iteration of the loop and starts the next iteration.
Note: Use `continue` statement instead of simpler `where` clause when you need a higher level of control.
Take the example of an 8 × 8 grid, wherein each cell holds a value of the row multiplied by the column:

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	8	10	12	14
3	0	3	6	9	12	15	18	21
4	0	4	8	12	16	20	24	28
5	0	5	10	15	20	25	30	35
6	0	6	12	18	24	30	36	42
7	0	7	14	21	28	35	42	49

It looks much like a multiplication table, doesn't it?
Let's say you wanted to calculate the sum of all cells but exclude all even rows, as shown below:

	0	1	2	3	4	5	6	7
0								
1	0	1	2	3	4	5	6	7
2								
3	0	3	6	9	12	15	18	21
4								
5	0	5	10	15	20	25	30	35
6								
7	0	7	14	21	28	35	42	49

Using a `for` loop, you can achieve this as follows:

```
sum = 0

for row in 0..<8 {
  if row % 2 == 0 {
    continue
  }
```

```
  for column in 0..<8 {
    sum += row * column
  }
}
```

When the row modulo `2` equals `0`, the row is even. In this case, `continue` makes the `for` loop skip to the next row.
Just like `break`, `continue` works with both `for` loops and `while` loops.
Note: Swift also provides *labeled statements* as location specifiers for `continue` and `break` in flow control elements. If you want to learn more, read our book Swift Apprentice, Fourth Edition.

Switch Statements

Another way to control flow is through the use of a `switch` statement, which lets you execute different bits of code depending on the value of a variable or constant.
Here's a very simple `switch` statement that acts on an integer:

```
let number = 10

switch number {
case 0:
  print("Zero")
default:
  print("Non-zero")
}
```

The code will print the following:
Non-zero
The purpose of this `switch` statement is to determine whether or not a number is zero.
To handle a specific case, you use `case`, followed by the value that you want to check for, which, in this case, is `0`. Then, you use `default` to signify what should happen for all other values.
Here's another example:

```
switch number {
case 10:
```

```
  print("It's ten!")
default:
  break
}
```

This time you check for 10, in which case, you print a message. Nothing should happen for other values. When you want nothing to happen for a case, or you want the default state to run, you use the `break` statement. This tells Swift that you meant to not write any code here and that nothing should happen. Cases can never be empty, so you must write some code, even if it's just a `break`!
`switch` statements work with any data type! Here's an example of switching on a string:

```
let string = "Dog"

switch string {
case "Cat", "Dog":
  print("Animal is a house pet.")
default:
  print("Animal is not a house pet.")
}
```

This will print the following:
Animal is a house pet.
In this example, you provide two values for the case, meaning that if the value is equal to either `"Cat"` or `"Dog"` then the statement will execute the case.

Advanced Switch Statements

You can also give your `switch` statements more than one case. In the previous section, you saw an `if` statement using multiple `else-if` statements to convert an hour of the day to a string describing that part of the day. You could rewrite that more succinctly with a `switch` statement, like so:

```
let hourOfDay = 12
var timeOfDay = ""

switch hourOfDay {
case 0, 1, 2, 3, 4, 5:
```

```
  timeOfDay = "Early morning"
case 6, 7, 8, 9, 10, 11:
  timeOfDay = "Morning"
case 12, 13, 14, 15, 16:
  timeOfDay = "Afternoon"
case 17, 18, 19:
  timeOfDay = "Evening"
case 20, 21, 22, 23:
  timeOfDay = "Late evening"
default:
  timeOfDay = "INVALID HOUR!"
}

print(timeOfDay)
```

This code will print the following:
Afternoon
Remember ranges? Well, you can use ranges to simplify this `switch` statement. You can rewrite it in a more succinct and clearer way using ranges as shown below:

```
var timeOfDay2 = ""

switch hourOfDay {
case 0...5:
  timeOfDay2 = "Early morning"
case 6...11:
  timeOfDay2 = "Morning"
case 12...16:
  timeOfDay2 = "Afternoon"
case 17...19:
  timeOfDay2 = "Evening"
case 20..<24:
  timeOfDay2 = "Late evening"
default:
```

```
  timeOfDay2 = "INVALID HOUR!"
}

print(timeOfDay2)
```

It's also possible to match a case to a condition based on a property of the value. As you learned in the first part of this tutorial series, you can use the modulo operator to determine if an integer is even or odd. Consider this code:

```
switch number {
case let x where x % 2 == 0:
  print("Even")
default:
  print("Odd")
}
```

This will print the following:
Even
This `switch` statement uses the `let-where` syntax, meaning the case will match only when a certain condition is `true`. The `let` part binds a value to a name, while the `where` part provides a Boolean condition that must be `true` for the case to match. In this example, you've designed the case to match if the value is even — that is, if the value modulo `2` equals `0`.
The method by which you can match values based on conditions is known as *pattern matching*.
In the previous example, the binding introduced an unnecessary constant `x`; it's simply another name for `number`. You are allowed to use `number` in the `where` clause and replace the binding with an underscore to ignore it:

```
switch number {
case _ where number % 2 == 0:
  print("Even")
default:
  print("Odd")
}
```

Partial Matching

Another way you can use `switch` statements with matching to great effect is as follows:

```
let coordinates = (x: 3, y: 2, z: 5)

switch coordinates {
case (0, 0, 0): // 1
  print("Origin")
case (_, 0, 0): // 2
  print("On the x-axis.")
case (0, _, 0): // 3
  print("On the y-axis.")
case (0, 0, _): // 4
  print("On the z-axis.")
default:        // 5
  print("Somewhere in space")
}
```

This `switch` statement makes use of *partial matching*. Here's what each case does, in order:

. Matches precisely the case in which the value is `(0, 0, 0)`. This is the origin of 3D space.

. Matches y=0, z=0 and any value of x. This means the coordinate is on the x-axis.

. Matches x=0, z=0 and any value of y. This means the coordinate is on the y-axis.

. Matches x=0, y=0 and any value of z. This means the coordinate is on the z-axis.

. Matches the remainder of coordinates.

You're using the underscore to mean that you don't care about the value. If you don't want to ignore the value, then you can bind it and use it in your `switch` statement, like this:

```
switch coordinates {
case (0, 0, 0):
  print("Origin")
case (let x, 0, 0):
  print("On the x-axis at x = \(x)")
case (0, let y, 0):
  print("On the y-axis at y = \(y)")
```

```
case (0, 0, let z):
  print("On the z-axis at z = \(z)")
case let (x, y, z):
  print("Somewhere in space at x = \(x), y = \(y), z = \(z)")
}
```

Here, the axis cases use the `let` syntax to pull out the pertinent values. The code then prints the values using string interpolation to build the string.
Notice how you don't need a default in this `switch` statement. This is because the final case is essentially the default; it matches anything, because there are no constraints on any part of the tuple. If the `switch` statement exhausts all possible values with its cases, then no default is necessary.
Also notice how you could use a single `let` to bind all values of the tuple: `let (x, y, z)` is the same as `(let x, let y, let z)`.
Finally, you can use the same `let-where` syntax that you saw earlier to match more complex cases. For example:

```
switch coordinates {
case let (x, y, _) where y == x:
  print("Along the y = x line.")
case let (x, y, _) where y == x * x:
  print("Along the y = x^2 line.")
default:
  break
}
```

Here, you match the "y equals x" and "y equals x squared" lines.

Where to Go From Here?

You can download the final playground using the *Download Materials* button at the top or bottom of this tutorial. To improve your Swift skills, you will find some mini-exercises to complete. If you are stuck or you need some help, feel free to take advantage of companion solutions.
You've learned a lot about the core language features for dealing with data over these past few tutorials — from data types to variables, then on to decision making with Booleans and loops with ranges.
If you have any questions or comments, please tell us in the discussion below!

This tutorial was taken from Chapters 4 and 5 of Swift Apprentice, Fourth Edition, available from the raywenderlich.com store.
Check it out and let us know what you think!

raywenderlich.com Weekly

The raywenderlich.com newsletter is the easiest way to stay up-to-date on everything you need to know as a mobile developer.

stevewozniak@apple.co

Get a weekly digest of our tutorials and courses, and receive a free in-depth email course as a bonus!

Average Rating

5/5

Add a rating for this content

Sign in to add a rating
12 ratings

Become a raywenderlich.com subscriber today!

Get immediate access to the highest-quality mobile development courses on the internet. With easy month-to-month subscriptions, learn Android, Kotlin, Swift & iOS development the easy way!