iOS Development: Lecture 2
SWIFT BASICS
DigiPAKISTAN - iOS Development Course

Prepared by:
Farhaj Ahmed, iOS Lead Trainer, DigiPAKISTAN

OPERATORS IN SWIFT

- COMPARISON OPERATOR
- COMPOUND COMPARISON OPERATOR
- NEGATION OPERATOR

OPERATORS IN SWIFT

COMPARISON OPERATOR

Comparison

The three main comparisons are "greater than" (>), "less than" (<), and "equals" (==). For exampe:

```
3 > 1    (true)
3 < 1    (false)
3 == 1   (false)
3 == 3   (true)
```

There is a special operator "not" (!) which is kind of like mathematical sarcasm (e.g. "Justin Bieber is cool... Not!). These operators can be combined to create three additional operators "greater than or equals" (>=), "less than or equals" (<=), and "not equals" (!=):

```
3 >= 4   // false
3 >= 3   // true
3 != 4   // true
3 != 3   // false
3  <= 4  // true
3 <= 3  // true
```

OPERATORS IN SWIFT

COMPOUND COMPARISON OPERATOR

Compound Comparison

All of the examples above have been simple expressions. We can form complex boolean expressions, that is expressions which are the result of more than one comparison, with the logical operators "and" (&&) and "or" (||). For example:

```
(3 >= 3)  &&  (3 <= 4)   // true
(3 >= 3)  ||  (3 <= 4)   // true
(3 >= 3)  &&  (3 > 4)   // false
```

Here's another example:

```
var providedPassword = true
var passedRetinaScan = false

providedPassword && passedRetinaScan  // false
providedPassword || passedRetinaScan // true


passedRetinaScan = true
providedPassword && passedRetinaScan // true
```

OPERATORS IN SWIFT

NEGATION OPERATOR

Negation

You can use the "not" (!) operator to invert the value of any boolean value or expression. Note that you can't have any space between the "!" and the value it's negating. For example:

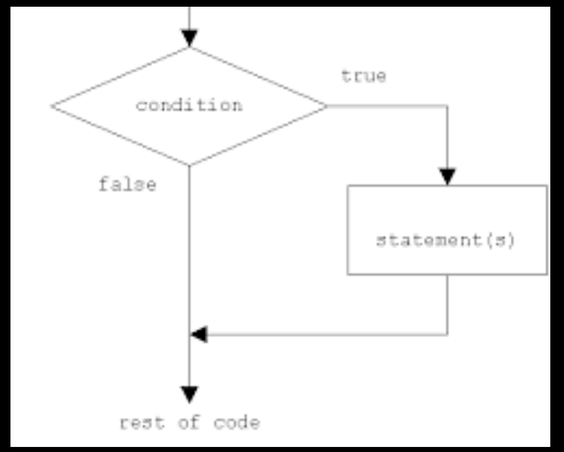
```
! true  (syntax error)
!true   (false)
!false   (true)
!(3 < 4)   (false)
!(3 > 4)   (true)
!((3 >= 3)  &&  (3 <= 4))   (false)
!((3 >= 3)  ||  (3 <= 4) )  (false)
!((3 >= 3)  &&  (3 > 4))   (true)
```

You can mix-and-match these operators in all kinds of different ways:

```
(3 < 4)  &&  !(3 <= 4)   (false)
```

WHAT IS A CONDITIONAL STATEMENT?

```
if 2 > 1 {
    print("Two is greater than One")
}
```



- A **conditional statement** is a set of rules performed if a certain condition is met. It is sometimes referred to as an **If-Then** statement, because **IF** a condition is met, THEN an action is performed.

HOW IF STATEMENT WORKS?

Swift if (if-then) Statement

The syntax of if statement in Swift is:

```
if expression {
        // statements
}
```

- Here `expression` is a boolean expression (returns either `true` or `false`).
- If the `expression` is evaluated to `true`, statements inside the code block of `if` is executed.
- If the `expression` is evaluated to `false`, statements inside the code block of `if` are skipped from execution.

HOW IF STATEMENT WORKS?

How if statement works?

Working of if statement when test condition is true

```
let test = 5

if test < 10 {
    // some code
    // some code
}

// statement just below if
```

Working of if statement when test condition is false

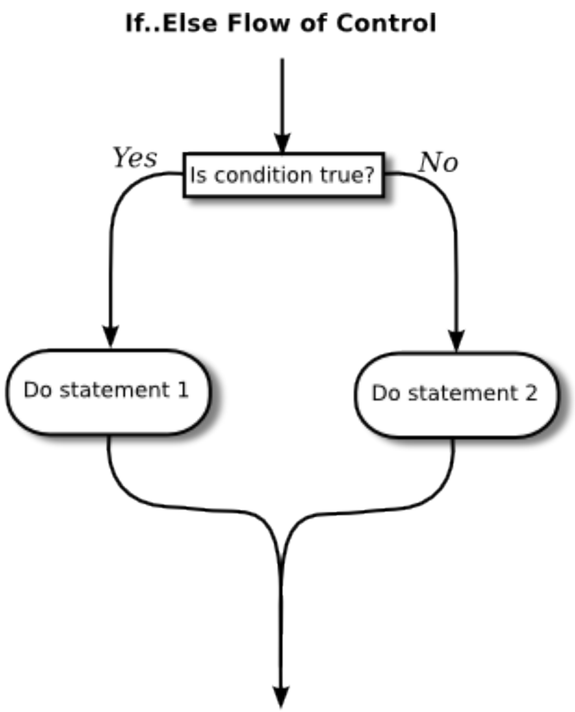
```
let test = 5

if test > 10 {
    // some code
    // some code
}

// statement just below if
```

Swift if statement working

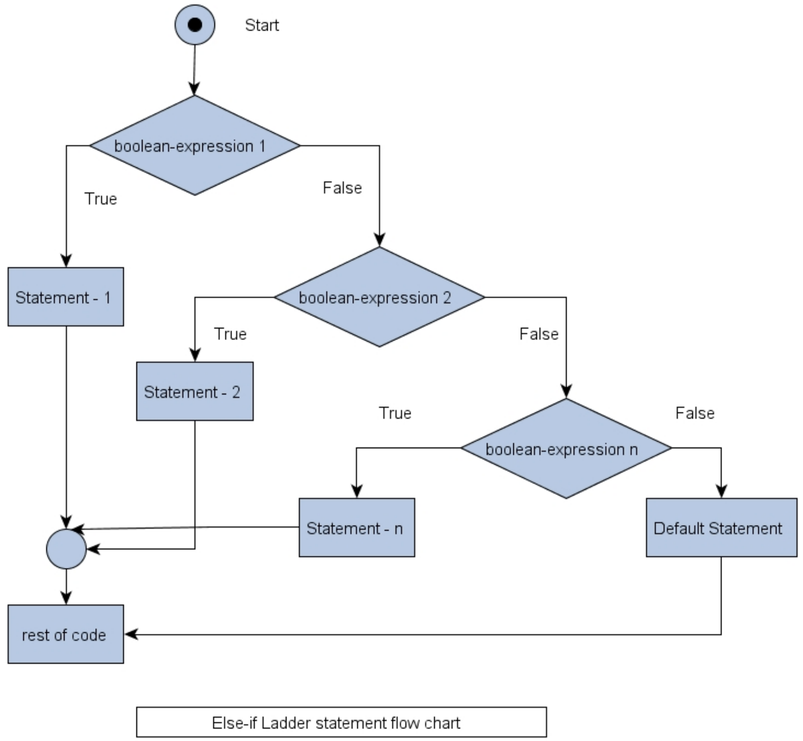
DigiPAKISTAN
Towards Digital Pakistan

ENOUGH TALK, LET'S JUMP TO <CODE/>

CONTROL FLOW



CONTROL FLOW



Else-if Ladder statement flow chart

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>
```

You can use this operator to rewrite your long code block above, like so:

```
let a = 5
let b = 10

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

In the first example, the condition is `a < b`. If this is true, the result assigned back to `min` will be the value of `a`; if it's false, the result will be the value of `b`.

I'm sure you'll agree that's much simpler! This is a useful operator that you'll find yourself using regularly.

Note: Because finding the greater or smaller of two numbers is such a common operation, the Swift standard library provides two functions for this purpose: `max` and `min`. If you were paying attention earlier in the book, then you'll recall you've already seen these.

BASIC CONTROL FLOW

If else Statement in Swift

Encapsulating variables

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

Let's take an example. Imagine 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
```

BASIC CONTROL FLOW

If else Statement in Swift

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

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**. Here's an example:

```
let animal = "Fox"

if animal == "Cat" || animal == "Dog" {
  print("Animal is a house pet.")
} else {
  print("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
let timeOfDay: String

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)
```

PRACTICE QUESTIONS

Create a constant named `myAge` and initialize it with your age. Write an `if` statement to print out `Teenager` if your age is between 13 and 19, and `Not a teenager` if your age is not between 13 and 19.

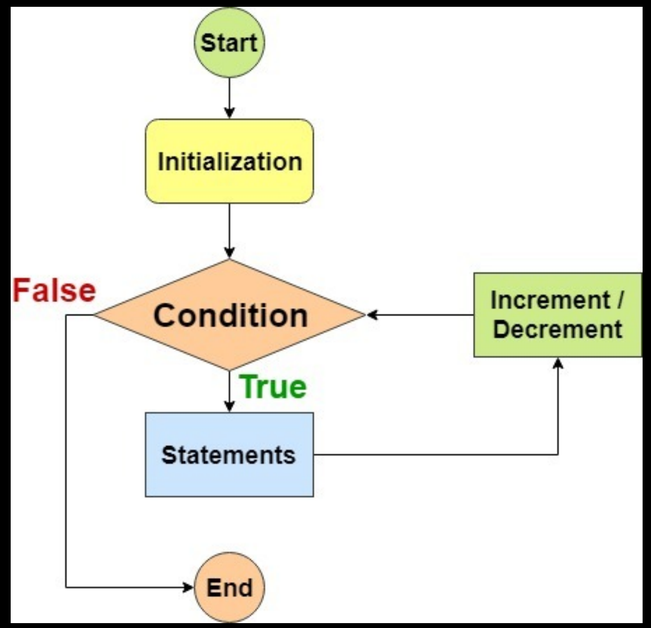
Create a constant named `answer` and use a ternary condition to set it equal to the result you print out for the same cases in the above exercise. Then print out `answer`.

What's wrong with the following code?

```
let firstName = "Matt"

if firstName == "Matt" {
  let lastName = "Galloway"
} else if firstName == "Ray" {
  let lastName = "Wenderlich"
}
let fullName = firstName + " " + lastName
```

WHAT ARE LOOPS?

- **Loops** are control structures used to **repeat** a given section of code a certain number of times or until a particular condition is met.
- A computer programmer who needs to use the same lines of code many times in a program can use a **loop** to **save time**.
- Loops are Swift's way of executing code **multiple times**.



SWIFT LOOPS

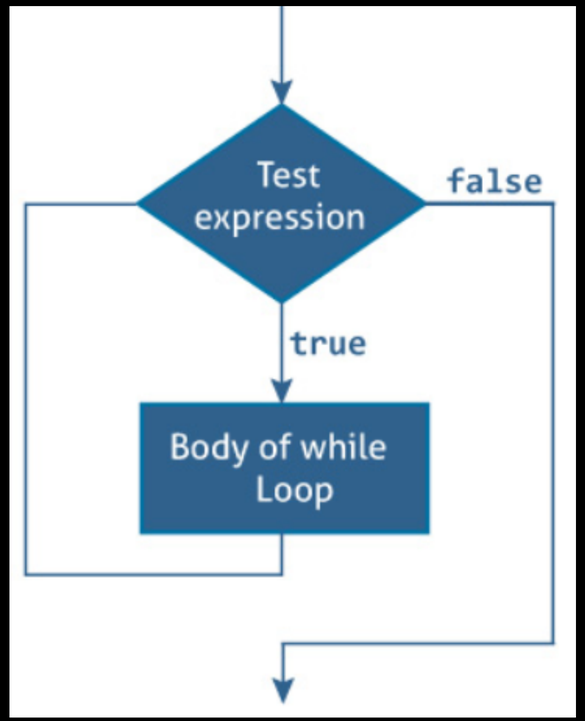
- WHILE LOOP
- REPEAT WHILE LOOP
- FOR LOOP

THE WHILE LOOP

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

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

- A **while loop** executes a set of statements until a condition becomes **false**. These kinds of **loops** are best used when the number of iterations is not known before the first iteration begins.



THE REPEAT WHILE LOOP

The body of **repeat...while** loop is executed once (before checking the test expression). Only then, **testExpression** is checked.

If **testExpression** is evaluated to **true**, statements inside the body of the loop are executed, and testExpression is evaluated again.

This process goes on until **testExpression** is evaluated to **false.** When testExpression is false, the **repeat..while loop** terminates.

```
repeat {
    // statement # 1
    // statement # 2
} while (testExpression)
```

BASIC CONTROL FLOW

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:

```
var 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:

```
var sum = 1

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

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

```
var 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.

BASIC CONTROL FLOW

Loops:

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:

```
var 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`. Neat!

You've seen how to write the same loop in different ways, demonstrating that in computer programming, there are often many ways to achieve the same result. You should choose the method that's easiest to read and conveys your intent in the best way possible. This is an approach you'll internalize with enough time and practice.

BASIC CONTROL FLOW

Summary:

- You use the Boolean data type `Bool` to represent true and false.
- The comparison operators, all of which return a Boolean, are:

```
Equal: ==
```

```
Not equal: !=
Less than: <
Greater than: >
Less than or equal: <=
Greater than or equal: >=
```

- You can use Boolean logic to combine comparison conditions.
- Swift's use of **canonicalization** ensures that the comparison of strings accounts for combining characters.
- You use `if` statements to make simple decisions based on a condition.
- You use `else` and `else-if` within an `if` statement to extend the decision-making beyond a single condition.
- Short circuiting ensures that only the minimal required parts of a Boolean expression are evaluated.
- You can use the ternary operator in place of simple `if` statements.
- Variables and constants belong to a certain scope, beyond which you cannot use them. A scope inherits visible variables and constants from its parent.
- While loops allow you to perform a certain task a number of times until a condition is met.
- The break statement lets you break out of a loop.

WHILE LOOP

The TestExpression is a boolean expression.
If the TestExpression is evaluated to true,

- statements inside the while loop are executed.
- and the TestExpression is evaluated again.

```
while (testExpression) {
    // statement # 1
    // statement # 2
}
```

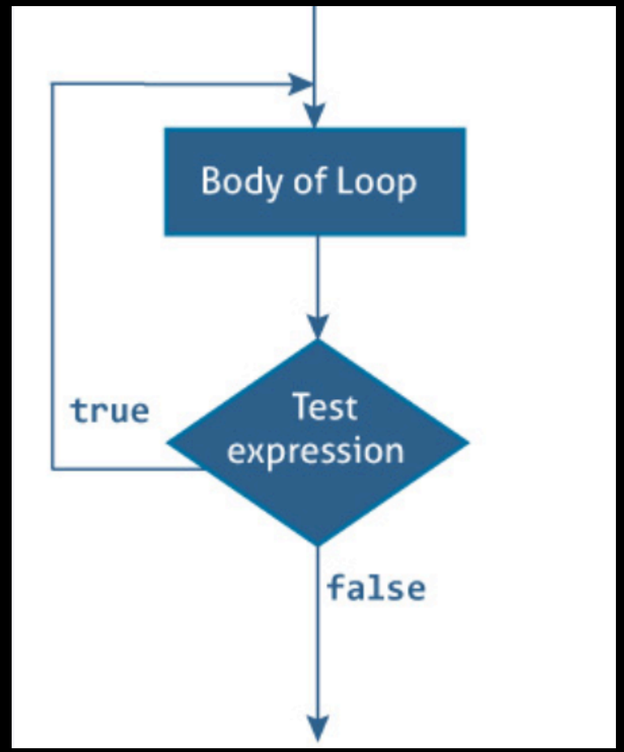
This process goes on until the TestExpression is evaluated to false. If the TestExpression evaluates to false, while loop is terminated.

THE REPEAT WHILE LOOP

- 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>
```

- This **loop** evaluates its condition at the end of each pass through the **loop**.The repeat...while **loop** is similar to while loop with one key difference. The body of **repeat...while loop** is executed once before the test expression is checked.



THE REPEAT WHILE LOOP

The body of **repeat...while** loop is executed once (before checking the test expression). Only then, **testExpression** is checked.

If **testExpression** is evaluated to **true**, statements inside the body of the loop are executed, and testExpression is evaluated again.

This process goes on until **testExpression** is evaluated to **false.** When testExpression is false, the **repeat..while loop** terminates.

```
repeat {
    // statement # 1
    // statement # 2
} while (testExpression)
```

PRACTICE QUESTIONS

1. Create a variable named `counter` and set it equal to `0`. Create a while loop with the condition `counter < 10` which prints out `counter is X` (where `X` is replaced with `counter` value) and then increments `counter` by `1`.

2. Create a variable named `counter` and set it equal to `0`. Create another variable named `roll` and set it equal to `0`. Create a `repeat-while` loop. Inside the loop, set `roll` equal to `Int.random(in: 0...5)` which means to pick a random number between `0` and `5`. Then increment `counter` by `1`. Finally, print `After X rolls, roll is Y` where `X` is the value of `counter` and `Y` is the value of `roll`. Set the loop condition such that the loop finishes when the first `0` is rolled.

PRACTICE QUESTIONS

Imagine you're playing a game of snakes & ladders that goes from position 1 to position 20. On it, there are ladders at position 3 and 7 which take you to 15 and 12 respectively. Then there are snakes at positions 11 and 17 which take you to 2 and 9 respectively.

Create a constant called `currentPosition` which you can set to whatever position between 1 and 20 which you like. Then create a constant called `diceRoll` which you can set to whatever roll of the dice you want. Finally, calculate the final position taking into account the ladders and snakes, calling it `nextPosition`.

Given a month (represented with a `String` in all lowercase) and the current year (represented with an `Int`), calculate the number of days in the month. Remember that because of leap years, "february" has 29 days when the year is a multiple of 4 but not a multiple of 100. February also has 29 days when the year is a multiple of 400.

PRACTICE QUESTIONS

Given a month (represented with a `String` in all lowercase) and the current year (represented with an `Int`), calculate the number of days in the month. Remember that because of leap years, "february" has 29 days when the year is a multiple of 4 but not a multiple of 100. February also has 29 days when the year is a multiple of 400.

Given a number, determine the next power of two above or equal to that number.

Given a number, print the triangular number of that depth. You can get a refresher of triangular numbers here: https://en.wikipedia.org/wiki/Triangular_number

Calculate the n'th Fibonacci number. Remember that Fibonacci numbers start its sequence with 1 and 1, and then subsequent numbers in the sequence are equal to the previous two values added together. You can get a refresher here: https://en.wikipedia.org/wiki/Fibonacci_number

CONCEPT OF RANGES

Before you dive into the `for` loop statement, you need to know about the **Countable Range** data types, which let you represent a sequence of countable integers. Let's look at two types of ranges.

First, there's **countable 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's the numbers `(0, 1, 2, 3, 4, 5)`.

Second, there's **countable 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 up to, but not including, 5. That's the numbers `(0, 1, 2, 3, 4)`.

Both open and half-open ranges must always be increasing. In other words, the second number must always be greater than or equal to the first. Countable ranges are commonly used in both `for` loops and `switch` statements, which means that throughout the rest of the chapter, you'll use ranges as well!

THE FOR LOOP

FOR LOOP

You construct a `for` loop like this:

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

The loop begins with the `for` keyword, followed by a name given to the loop constant (more on that shortly), 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
}
```

In the code above, the `for` loop iterates through the range 1 to `count`. At the first iteration, `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.

DigiPAKISTAN
Towards Digital Pakistan

ENOUGH TALK, LET'S JUMP TO <CODE/>

PRACTICE QUESTIONS

1. Create a constant named `range`, and set it equal to a range starting at 1 and ending with 10 inclusive. Write a `for` loop that iterates over this range and prints the square of each number.

2. Write a `for` loop to iterate over the same range as in the exercise above and print the square root of each number. You'll need to type convert your loop constant.

3. Above, you saw a `for` loop that iterated over only the even rows like so:

```
sum = 0
for row in 0..<8 {
  if row % 2 == 0 {
    continue
  }
  for column in 0..<8 {
    sum += row * column
  }
}
```

Change this to use a `where` clause on the first `for` loop to skip even rows instead of using `continue`. Check that the sum is 448 as in the initial example.

PRACTICE QUESTIONS

In the following `for` loop, what will be the value of `sum`, and how many iterations will happen?

```
var sum = 0
for i in 0...5 {
  sum += i
}
```

In the `while` loop below, how many instances of "a" will there be in `aLotOfAs`? Hint: `aLotOfAs.count` tells you how many characters are in the string `aLotOfAs`.

```
var aLotOfAs = ""
while aLotOfAs.count < 10 {
  aLotOfAs += "a"
}
```

SWITCH STATEMENTS

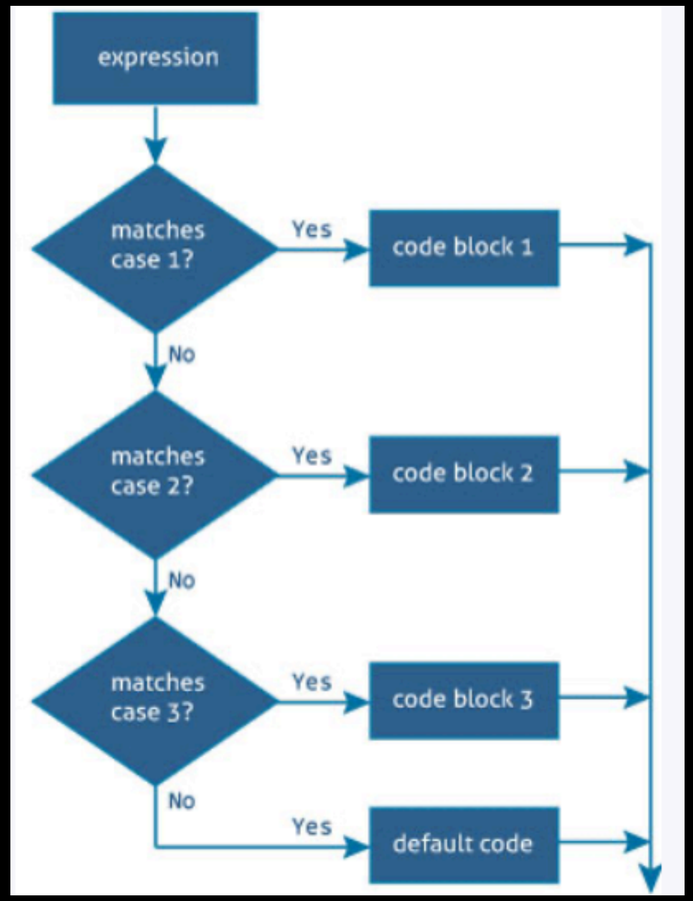
```
let string = "Dog"

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

- **Swift** has another type of flow control called **switch/case**. It's easiest to think of this as being an advanced form of **if**, because you can have lots of matches and Swift will execute the right one.

- In the most basic form of a **switch/case** you tell **Swift** what variable you want to check, then provide a list of possible cases for that variable. Swift will find the first case that matches your variable, then run its block of code. When that block finishes, Swift exits the whole **switch/case** block.

HOW SWITCH STATEMENTS WORKS?

- The **switch** expression is evaluated once.
- It takes the expression and compares with each case value in the order **(Top -> Bottom)**.
- If there is a match, the statement inside the case are executed and the entire switch statement finishes its execution as soon as the **first matching switch** case is completed.
- If there is no match for the case, it falls to the next case.
- The **default** keyword specifies the code to run if there is no case match.



DigiPAKISTAN
Towards Digital Pakistan

ENOUGH TALK, LET'S JUMP TO <CODE/>

ADVANCED CONTROL FLOW

Switch Statements

Of course, `switch` statements also work with data types other than integers. They 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 CONTROL FLOW

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")
}
```

In this example, the code will print the following:

```
Non-zero
```

Here's another example:

```
let number = 10

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`!

ADVANCED CONTROL FLOW

Switch Statements

You can also give your `switch` statements more than one case. In the previous chapter, 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
let timeOfDay: String

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 using ranges as shown below:

```
let hourOfDay = 12
let timeOfDay: String

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

ADVANCED CONTROL FLOW

Conditions in Switch Statements

```
let number = 10

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.

In the previous example, the binding introduced a 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:

```
let number = 10

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

ADVANCED CONTROL FLOW

Tuples in Switch Statements

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:

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

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)`.

ADVANCED CONTROL FLOW

Tuples in Switch Statements

Finally, you can use the same `let-where` syntax you saw earlier to match more complex cases. For example:

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

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.

And those are the basics of `switch` statements!

ADVANCED CONTROL FLOW

Summary

- You can use **ranges** to create a sequence of numbers, incrementing to move from one value to another.
- **Closed ranges** include both the start and end values.
- **Half-open ranges** include the start value and stop one before the end value.
- **For loops** allow you to iterate over a range.
- The **continue** statement lets you finish the current iteration of a loop and begin the next iteration.
- **Labeled statements** let you use `break` and `continue` on an outer loop.
- You use `switch` statements to decide which code to run depending on the value of a variable or constant.
- The power of a `switch` statement comes from leveraging pattern matching to compare values using complex rules.

PRACTICE QUESTIONS

Consider the following `switch` statement:

```
switch coordinates {
case let (x, y, z) where x == y && y == z:
  print("x = y = z")
case (_, _, 0):
  print("On the x/y plane")
case (_, 0, _):
  print("On the x/z plane")
case (0, _, _):
  print("On the y/z plane")
default:
  print("Nothing special")
}
```

What will this code print when `coordinates` is each of the following?

```
let coordinates = (1, 5, 0)
let coordinates = (2, 2, 2)
let coordinates = (3, 0, 1)
let coordinates = (3, 2, 5)
let coordinates = (0, 2, 4)
```

A closed range can never be empty. Why?

Print a countdown from 10 to 0. (Note: do not use the `reversed()` method, which will be introduced later.)

Print 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0. (Note: do not use the `stride(from:by:to:)` function, which will be introduced later.)

PRACTICE QUESTIONS

Write a switch statement that takes an age as an integer and prints out the life stage related to that age. You can make up the life stages, or use my categorization as follows: 0-2 years, Infant; 3-12 years, Child; 13-19 years, Teenager; 20-39, Adult; 40-60, Middle aged; 61+, Elderly.

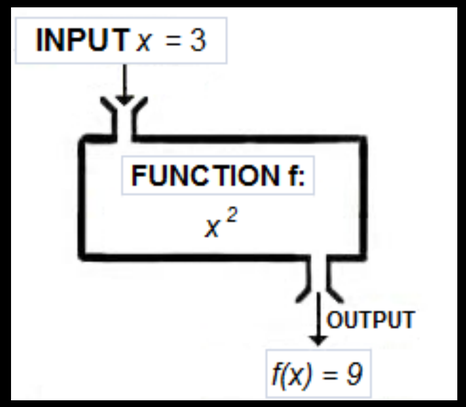
Write a `switch` statement that takes a tuple containing a string and an integer. The string is a name, and the integer is an age. Use the same cases that you used in the previous exercise and `let` syntax to print out the name followed by the life stage. For example, for myself it would print out `"Matt is an adult."`.

ADVANCED CONTROL FLOW

Summary

- You can use **ranges** to create a sequence of numbers, incrementing to move from one value to another.
- **Closed ranges** include both the start and end values.
- **Half-open ranges** include the start value and stop one before the end value.
- **For loops** allow you to iterate over a range.
- The **continue** statement lets you finish the current iteration of a loop and begin the next iteration.
- **Labeled statements** let you use `break` and `continue` on an outer loop.
- You use `switch` statements to decide which code to run depending on the value of a variable or constant.
- The power of a `switch` statement comes from leveraging pattern matching to compare values using complex rules.

CONCEPT OF FUNCTIONS



- ***Functions*** are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to **“call”** the function to perform its task when needed.

- Every **function** in **Swift** has a type, consisting of the function’s **parameter** types and **return** type. You can use this type like any other type in Swift, which makes it easy to pass functions as parameters to other functions, and to return functions from functions.

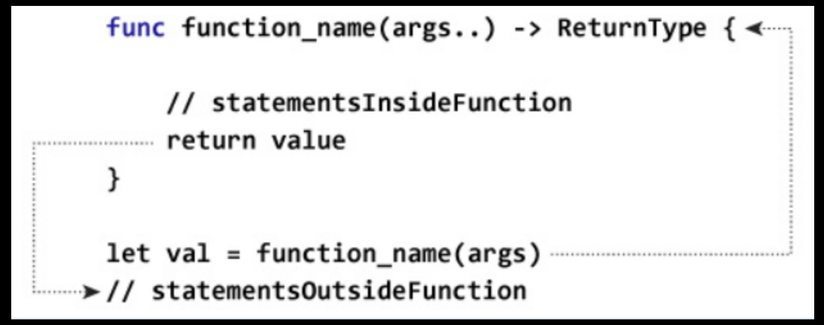
DigiPAKISTAN
Towards Digital Pakistan

ENOUGH TALK, LET'S JUMP TO <CODE/>

HOW FUNCTION WORKS?

The statement **function_name(args)** invokes/calls the function with argument values args, which then leaves the current section of code (i.e. stops executing statements below it) and begins to execute the first line inside the function.

1. The program comes to a line of code func **function_name(Args...)** and accepts the values args passed during the function call function_name(args).
2. The program then executes the statements statements Inside **Function** defined inside the function.
3. The statements inside the function are executed in **top to bottom order**, one after the other.
4. After the execution of the last statement, the program leaves the function and goes back to where it started from i.e **function_name(args)**.
5. let **val = stores** the value returned from the function in a constant val. Similarly, you can store in a variable as var val =.
6. After that, statements statementsOutsideFunction are executed.In the above diagram, the statement **function_name(args)** invokes/calls the function with argument values args, which then leaves the current section of code (i.e. stops executing statements below it) and begins to execute the first line inside the function.
7. The program comes to a line of code func **function_name(Args...)** and accepts the values args passed during the function call **function_name(args)**.
8. The program then executes the statements statementsInsideFunction defined inside the function.
9. The statements inside the function are executed in top to bottom order, one after the other.
10. After the execution of the last statement, the program leaves the function and goes back to where it started from i.e **function_name(args)**.
11. let **val = stores** the value returned from the function in a constant val. Similarly, you can store in a variable as var val =.
12. After that, statements **statementsOutsideFunction** are executed.



FUNCTIONS

Function Basics

Say you have an app that often needs to print your name. You can write a function to do this:

```
func printMyName() {
  print("My name is Matt Galloway.")
}
```

The code above is known as a **function declaration**. You define a function using the `func` keyword. After that comes the name of the function, followed by parentheses. You'll learn more about the need for these parentheses in the next section.

After the parentheses comes an opening brace, followed by the code you want to run in the function, followed by a closing brace. With your function defined, you can use it like so:

```
printMyName()
```

This prints out the following:

```
My name is Matt Galloway.
```

FUNCTIONS

Function Basics

Function parameters

In the previous example, the function simply prints out a message. That's great, but sometimes you want to **parameterize** your function, which lets the function perform differently depending on the data passed into it via its **parameters**.

As an example, consider the following function:

```
func printMultipleOfFive(value: Int) {
  print("\(value) * 5 = \(value * 5)")
}
printMultipleOfFive(value: 10)
```

Here, you can see the definition of one parameter inside the parentheses after the function name, named `value` and of type `Int`. In any function, the parentheses contain what's known as the **parameter list**. The parentheses are required to invoke the function, even if the parameter list is empty.

This function will print out any given multiple of five. In the example, you call the function with an **argument** of 10, so the function prints the following:

```
10 * 5 = 50
```

FUNCTIONS

Function Basics

> **Note:** Take care not to confuse the terms "parameter" and "argument". A function declares its *parameters* in its parameter list. When you call a function, you provide values as *arguments* for the functions parameters.

You can take this one step further and make the function more general. With two parameters, the function can print out a multiple of any two values:

```
func printMultipleOf(multiplier: Int, andValue: Int) {
  print("\(multiplier) * \(andValue) = \(multiplier * andValue)")
}
printMultipleOf(multiplier: 4, andValue: 2)
```

There are now two parameters inside the parentheses after the function name: one named `multiplier` and the other named `andValue`, both of type `Int`.

Notice that you need to apply the labels in the parameter list to the arguments when you call a function. In the example above you need to put `multiplier:` before the multiplier and `andValue:` before the value to be multiplied.

In Swift, you should try to make your function calls read like a sentence. In the example above, you would read the last line of code like this:

> *Print multiple of multiplier 4 and value 2*

You can make this even clearer by giving a parameter a different external name. For example, you can change the name of the `andValue` parameter:

```
func printMultipleOf(multiplier: Int, and value: Int) {
  print("\(multiplier) * \(value) = \(multiplier * value)")
}
printMultipleOf(multiplier: 4, and: 2)
```

You assign a different external name by writing it in front of the parameter name. In this example, the internal name of the parameter is now `value` while the external name (the argument label) in the function call is now `and`. You can read the new call as:

> *Print multiple of multiplier 4 and 2*

FUNCTIONS

Function Basics

If you want to have no external name at all, then you can employ the underscore _, as you've seen in previous chapters:

```
func printMultipleOf(_ multiplier: Int, and value: Int) {
  print("\(multiplier) * \(value) = \(multiplier * value)")
}
printMultipleOf(4, and: 2)
```

This makes it even more readable. The function call now reads like so:

```
Print multiple of 4 and 2
```

You could, if you so wished, take this even further and use _ for all parameters, like so:

```
func printMultipleOf(_ multiplier: Int, _ value: Int) {
  print("\(multiplier) * \(value) = \(multiplier * value)")
}
printMultipleOf(4, 2)
```

In this example, all parameters have no external name. But this illustrates how you use the underscore wisely. Here, your expression is still understandable, but more complex functions that take many parameters can become confusing and unwieldy with no external parameter names. Imagine if a function took five parameters!

You can also give default values to parameters:

```
func printMultipleOf(_ multiplier: Int, _ value: Int = 1) {
  print("\(multiplier) * \(value) = \(multiplier * value)")
}
printMultipleOf(4)
```

The difference is the `= 1` after the second parameter, which means that if no value is provided for the second parameter, it defaults to 1.

Therefore, this code prints the following:

```
4 * 1 = 4
```

FUNCTIONS

Return Values and Returning Multiple Variables

All of the functions you've seen so far have performed a simple task, namely, printing out something. Functions can also return a value. The caller of the function can assign the return value to a variable or constant or use it directly in an expression.

This means you can use a function to manipulate data. You simply take in data through parameters, manipulate it and then return it. Here's how you define a function that returns a value:

```
func multiply(_ number: Int, by multiplier: Int) -> Int {
  return number * multiplier
}
let result = multiply(4, by: 2)
```

To declare that a function returns a value, after the set of parentheses and before the opening brace, you add a -> followed by the type of the return value. In this example, the function returns an `Int`.

Inside the function, you use a `return` statement to return the value. In this example, you return the product of the two parameters.

It's also possible to return multiple values through the use of tuples:

```
func multiplyAndDivide(_ number: Int, by factor: Int)
                    -> (product: Int, quotient: Int) {
  return (number * factor, number / factor)
}
let results = multiplyAndDivide(4, by: 2)
let product = results.product
let quotient = results.quotient
```

This function returns *both* the product and quotient of the two parameters; it returns a tuple containing two `Int` values with appropriate member value names.

The ability to return multiple values through tuples is one thing that makes it such a pleasure to work with Swift. And it turns out to be a very useful feature, as you'll see shortly.

FUNCTIONS

Advanced Parameter Handling

Function parameters are constants by default, which means they can't be modified. To illustrate this point, consider the following code:

```
func incrementAndPrint(_ value: Int) {
  value += 1
  print(value)
}
```

This results in an error:

```
Left side of mutating operator isn't mutable: 'value' is a 'let' constant
```

The parameter `value` is the equivalent of a constant declared with `let`. Therefore, when the function attempts to increment it, the compiler emits an error.

An important point to note is that Swift copies the value before passing it to the function, a behavior known as **pass-by-value**.

FUNCTIONS

Advanced Parameter Handling

Function parameters are constants by default, which means they can't be modified. To illustrate this point, consider the following code:

```
func incrementAndPrint(_ value: Int) {
  value += 1
  print(value)
}
```

This results in an error:

```
Left side of mutating operator isn't mutable: 'value' is a 'let' constant
```

The parameter `value` is the equivalent of a constant declared with `let`. Therefore, when the function attempts to increment it, the compiler emits an error.

An important point to note is that Swift copies the value before passing it to the function, a behavior known as **pass-by-value**.

Sometimes you *do* want to let a function change a parameter directly, a behavior known as **copy-in copy-out** or **call by value result**. You do it like so:

```
func incrementAndPrint(_ value: inout Int) {
  value += 1
  print(value)
}
```

The `inout` keyword before the parameter type indicates that this parameter should be copied in, that local copy used within the function, and then copied back out when the function returns.

You need to make a slight tweak to the function call to complete this example. Add an ampersand (`&`) before the argument, which makes it clear at the call site that you are using copy-in copy-out:

```
var value = 5
incrementAndPrint(&value)
print(value)
```

Now the function can change the value however it wishes.

FUNCTIONS

Function Overloading

Did you notice how you used the same function name for several different functions in the previous examples?

```
func printMultipleOf(multiplier: Int, andValue: Int)
func printMultipleOf(multiplier: Int, and value: Int)
func printMultipleOf(_ multiplier: Int, and value: Int)
func printMultipleOf(_ multiplier: Int, _ value: Int)
```

This technique is called **overloading** and allows you to define similar functions using a single name.

However, the compiler must still be able to tell the difference between these functions. Whenever you call a function, it should always be clear which function you're calling. This is usually achieved through a difference in the parameter list:

- A different number of parameters.
- Different parameter types.
- Different external parameter names, such as the case with `printMultipleOf`.

You can also overload a function name based on a different return type, like so:

```
func getValue() -> Int {
  return 31;
}

func getValue() -> String {
  return "Matt Galloway"
}
```

Here, there are two functions called `getValue()`, which return different types. One an `Int` and the other a `String`.

FUNCTIONS

Mini Exercises

1. Write a function named `printFullName` that takes two strings called `firstName` and `lastName`. The function should print out the full name defined as `firstName + " " + lastName`. Use it to print out your own full name.
2. Change the declaration of `printFullName` to have no external name for either parameter.
3. Write a function named `calculateFullName` that returns the full name as a string. Use it to store your own full name in a constant.
4. Change `calculateFullName` to return a tuple containing both the full name and the length of the name. You can find a string's length by using the following syntax: `string.characters.count`. Use this function to determine the length of your own full name.

FUNCTIONS

Functions as Variables

This may come as a surprise, but functions in Swift are simply another data type. You can assign them to variables and constants just as you can any other type of value, such as an `Int` or a `String`.

FUNCTIONS

Functions as Variables

To see how this works, consider the following function:

```
func add(_ a: Int, _ b: Int) -> Int {
  return a + b
}
```

This function takes two parameters and returns the sum of their values.

You can assign this function to a variable, like so:

```
var function = add
```

Here, the name of the variable is `function` and its type is inferred as `(Int, Int) -> Int` from the `add` function you assign to it.

Notice how the function type `(Int, Int) -> Int` is written in the same way you write the parameter list and return type in a function declaration. Here, the `function` variable is of a function type that takes two `Int` parameters and returns an `Int`.

Now you can use the `function` variable in just the same way you'd use `add`, like so:

```
function(4, 2)
```

This returns 6.

Now consider the following code:

```
func subtract(_ a: Int, _ b: Int) -> Int {
  return a - b
}
```

Here, you declare another function that takes two `Int` parameters and returns an `Int`. You can set the `function` variable from before to your new `subtract` function, because the parameter list and return type of `subtract` are compatible with the type of the `function` variable.

```
function = subtract
function(4, 2)
```

This time, the call to `function` returns 2.

The fact that you can assign functions to variables comes in handy because it means you can pass functions to other functions. Here's an example of this in action:

```
func printResult(_ function: (Int, Int) -> Int, _ a: Int, _ b: Int) {
  let result = function(a, b)
  print(result)
}
printResult(add, 4, 2)
```

`printResult` takes three parameters:

1. `function` is of a function type that takes two `Int` parameters and returns an `Int`, declared like so: `(Int, Int) -> Int`.
2. `a` is of type `Int`.
3. `b` is of type `Int`.

`printResult` calls the passed-in function, passing into it the two `Int` parameters. Then it prints the result to the console:

```
6
```

It's extremely useful to be able to pass functions to other functions, and it can help you write reusable code. Not only can you pass data around to manipulate, but passing functions as parameters lets you be flexible about what code gets executed too.

PRACTICE QUESTIONS

1. Write a function named `printFullName` that takes two strings called `firstName` and `lastName`. The function should print out the full name defined as `firstName + " " + lastName`. Use it to print out your own full name.
2. Change the declaration of `printFullName` to have no external name for either parameter.
3. Write a function named `calculateFullName` that returns the full name as a string. Use it to store your own full name in a constant.
4. Change `calculateFullName` to return a tuple containing both the full name and the length of the name. You can find a string's length by using the `count` property. Use this function to determine the length of your own full name.

WHAT IS FUNCTION OVERLOADING?

```
func getValue() -> Int {
  31
}

func getValue() -> String {
  "Matt Galloway"
}
```

- When two are more functions have same name but different arguments, they are known as overloaded functions and this process in known as function overloading.

This is usually achieved through a difference in the parameter list:

- A different number of parameters.
- Different parameter types.
- Different external parameter names.

HOW TO DOCUMENT YOUR FUNCTION?

Fortunately Swift has a very easy way to document functions which integrates well with Xcode's code completion and other features.

It uses the defacto **Doxygen** commenting standard used by many other languages outside of Swift. Let's take a look at how you can document a function:

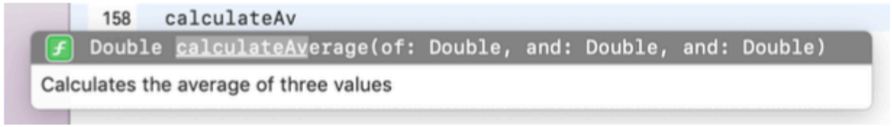
```
/// Calculates the average of three values
/// - Parameters:
///   - a: The first value.
///   - b: The second value.
///   - c: The third value.
/// - Returns: The average of the three values.
func calculateAverage(of a: Double, and b: Double, and c:
Double) -> Double {
  let total = a + b + c
  let average = total / 3
  return average
}
calculateAverage(of: 1, and: 3, and: 5)
```

Instead of the usual double-/, you use triple-/ instead. Then the first line is the description of what the function does. Following that is a list of the parameters and finally a description of the return value.

If you forget the format of a documentation comment, simply highlight the function and press "Option-Command-/" in Xcode. The Xcode editor will insert a comment template for you that you can then fill out.

When you create this kind of code documentation, you will find that the comment changes font in Xcode from the usual monospace font. Neat right? Well yes, but there's more.

First, your documentation is shown when code completion comes up, like so:



FUNCTIONS AS VARIABLES

This may come as a surprise, but functions in Swift are simply another data type. You can assign them to variables and constants just as you can any other type of value, such as an `Int` or a `String`.

To see how this works, consider the following function:

```
func add(_ a: Int, _ b: Int) -> Int {
  a + b
}
```

This function takes two parameters and returns the sum of their values.

You can assign this function to a variable, like so:

```
var function = add
```

Here, the name of the variable is `function` and its type is inferred as `(Int, Int) -> Int` from the `add` function you assign to it.

PRACTICE QUESTIONS

When I'm acquainting myself with a programming language, one of the first things I do is write a function to determine whether or not a number is prime. That's your second challenge.

First, write the following function:

```
func isNumberDivisible(_ number: Int, by divisor: Int) -> Bool
```

You'll use this to determine if one number is divisible by another. It should return `true` when `number` is divisible by `divisor`.

Hint: You can use the modulo (`%`) operator to help you out here.

Next, write the main function:

```
func isPrime(_ number: Int) -> Bool
```

This should return `true` if `number` is prime, and `false` otherwise. A number is prime if it's only divisible by 1 and itself. You should loop through the numbers from 1 to the number and find the number's divisors. If it has any divisors other than 1 and itself, then the number isn't prime. You'll need to use the `isNumberDivisible(_:by:)` function you wrote earlier.

Use this function to check the following cases:

```
isPrime(6) // false
isPrime(13) // true
isPrime(8893) // true
```

Hint 1: Numbers less than 0 should not be considered prime. Check for this case at the start of the function and return early if the number is less than 0.

Hint 2: Use a `for` loop to find divisors. If you start at 2 and end before the number itself, then as soon as you find a divisor, you can return `false`.

Hint 3: If you want to get *really* clever, you can simply loop from 2 until you reach the square root of `number`, rather than going all the way up to `number` itself. I'll leave it as an exercise for you to figure out why. It may help to think of the number 16, whose square root is 4. The divisors of 16 are 1, 2, 4, 8 and 16.

PRACTICE QUESTIONS

You're going to write a function that computes a value from the **Fibonacci sequence**. Any value in the sequence is the sum of the previous two values. The sequence is defined such that the first two values equal 1. That is, `fibonacci(1) = 1` and `fibonacci(2) = 1`.

Write your function using the following declaration:

```
func fibonacci(_ number: Int) -> Int
```

Then, verify you've written the function correctly by executing it with the following numbers:

```
fibonacci(1)  // = 1
fibonacci(2)  // = 1
fibonacci(3)  // = 2
fibonacci(4)  // = 3
fibonacci(5)  // = 5
fibonacci(10) // = 55
```

Hint 1: For values of `number` less than 0, you should return 0.

Hint 2: To start the sequence, hard-code a return value of 1 when `number` equals 1 or 2.

Hint 3: For any other value, you'll need to return the sum of calling `fibonacci` with `number - 1` and `number - 2`.

DigiPAKISTAN
Towards Digital Pakistan

YOU CAN FIND ME @:


farhaj.ahmed@live.com


https://www.linkedin.com/in/farhajahmed1


https://www.facebook.com/LearnFromFarhaj

DigiPAKISTAN
Towards Digital Pakistan

THANK YOU