iOS Development - Lecture 1
iOS Development Course by Farhaj Ahmed

Prepared by:
Farhaj Ahmed, iOS Lead Trainer, DigiPAKISTAN

Xcode – Tool for iOS Application Development

Xcode is an integrated development environment (IDE) for macOS containing suite of software development tools developed by Apple for developing software for macOS, iOS, watchOS, and tvOS.

First released in 2003, the latest stable release is version 12.5 and is available via the Mac App Store free of charge for macOS High Sierra and macOS Mojave users

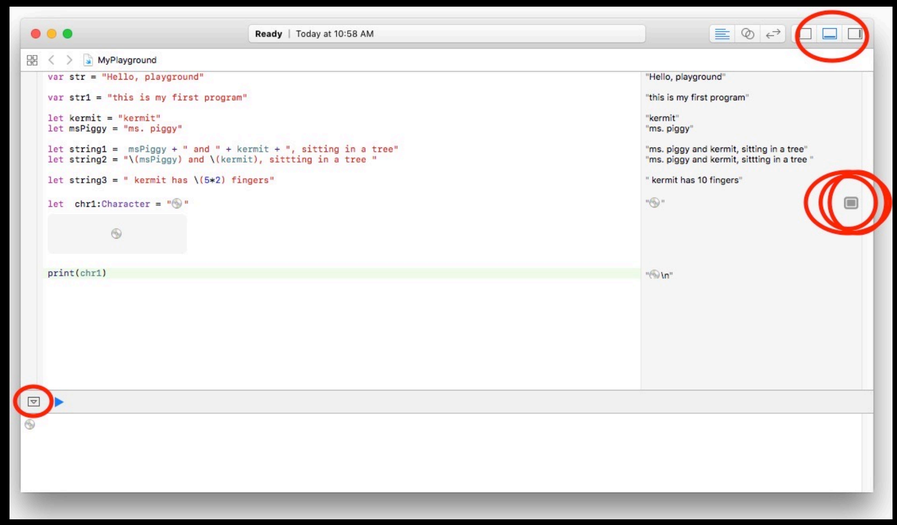
SWIFT PROGRAMMING LANGUAGE

SWIFT IS A GENERAL-PURPOSE, MULTI-PARADIGM, COMPILED PROGRAMMING LANGUAGE DEVELOPED BY APPLE INC. FOR IOS, MACOS, WATCHOS, TVOS, AND LINUX.

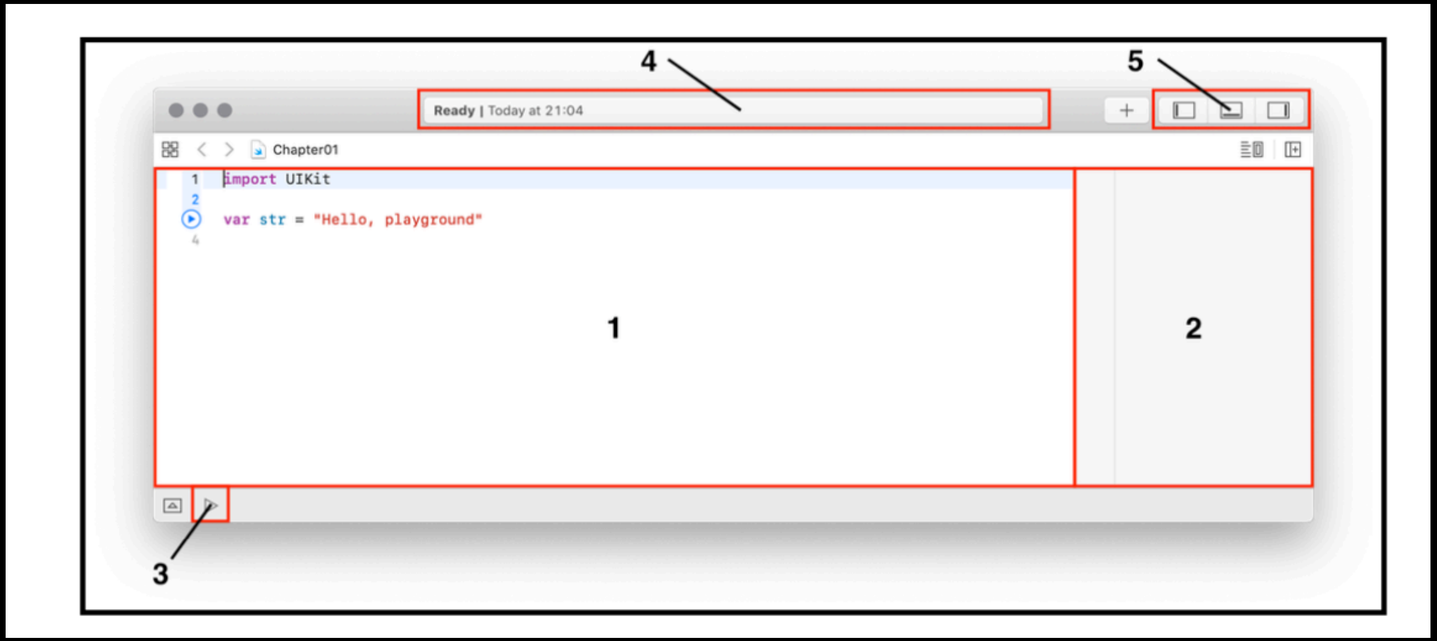
SWIFT WAS INTRODUCED AT APPLE'S 2014 WORLDWIDE DEVELOPERS CONFERENCE.

SWIFT Playground

- **In Swift Playgrounds you create small programs called “playgrounds” that instantly show the results of the code that you write**



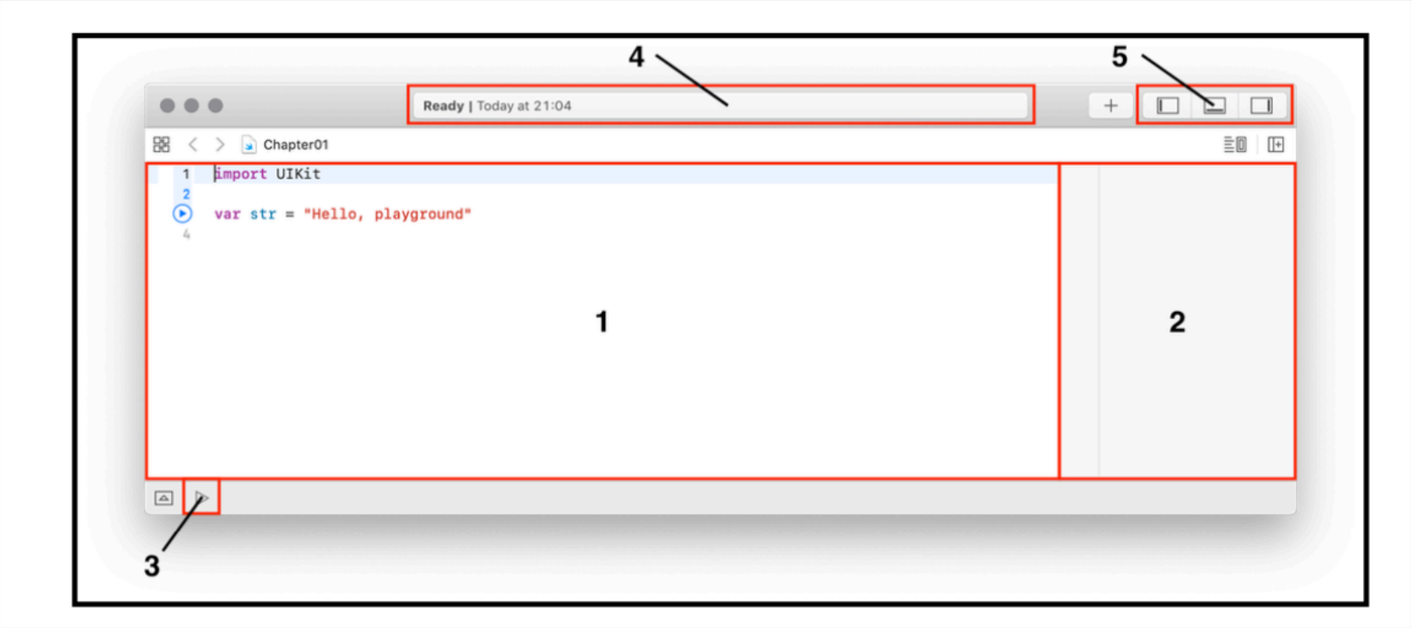
SWIFT Playground



1. Source editor: This is the area in which you'll write your Swift code. It's much like a text editor such as Notepad or TextEdit. You'll notice the use of what's known as a monospaced font, meaning all characters are the same width. This makes the code much easier to read and format.

2. Results sidebar: This area shows the results of your code. You'll learn more about how code is executed as you read through the book. The results sidebar will be the main place you'll look to confirm your code is working as expected

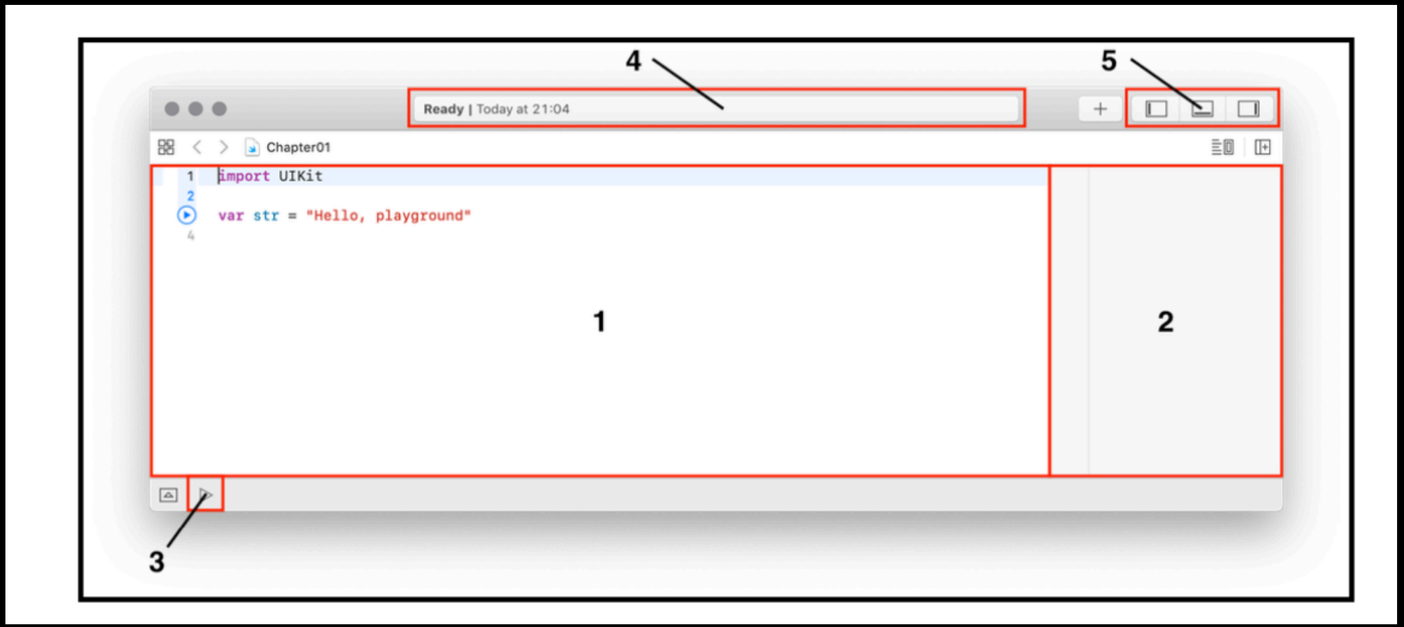
SWIFT Playground



3. **Execution control**: This control lets you run the entire playground file or clear state so you can run it again. By default, playgrounds do not execute automatically. You can change this setting to execute with every change by long pressing on it and selecting "Automatically Run".

4. **Activity viewer**: This shows the status of the playground. In the screenshot, it shows that the playground has finished executing and is ready to handle more code in the source editor. When the playground is executing, this viewer will indicate this with a spinner.

SWIFT Playground



5. Panel controls: These toggle switches show and hide three panels, one that appears on the left, one on the bottom and one on the right. The panels each display extra information that you may need to access from time to time. You'll usually keep them hidden, as they are in the screenshot. You'll learn more about each of these panels as you move through the book.

ONLINE COMPILERS TO RUN SWIFT 5

DigiPAKISTAN
Towards Digital Pakistan

1. JDOODLE: https://www.jdoodle.com/execute-swift-online/

2. iSwift: https://iswift.org/playground

3. Replit: - https://repl.it/languages/swift

4. Piazo: https://paiza.io/en/projects/new?language=swift

HOW TO SET UP MAC OS WITHOUT MAC?

DigiPAKISTAN
Towards Digital Pakistan

There are 3 ways to setup MacOS without Mac

1. Virtual Box Installation
2. VMWare Installation - Recommended
3. Hackintosh Installation

HOW TO SETUP MACOS WITHOUT MAC

DigiPAKISTAN
Towards Digital Pakistan

Links to Setup MacOS VMware Installation

For MacOS Big Sur

https://www.geekrar.com/install-macos-catalina-on-vmware-on-windows-pc/

ISO File Download Link:

https://drive.google.com/file/d/116Dm_X_QZJ5eGMuk7NVN_j2Rhk-oAZQB/view

https://drive.google.com/drive/folders/1rnNhFvhSRk9eCmQYgmBP0ihrPk2Iu9xr

For MacOS Catalina

https://www.geekrar.com/installation-of-macos-big-sur-11-0-guided-steps/

ISO File Download Link:

https://drive.google.com/drive/folders/1ZX- YEPPNC6aT_w8eiU3ztaLhXH6-e4zx

HOW TO SETUP MACOS WITHOUT MAC

DigiPAKISTAN
Towards Digital Pakistan

Links to Setup MacOS Virtual Box Installation:

For MacOS Big Sur

https://techrechard.com/how-to-install-macos-big-sur-on-virtualbox-on-windows/

For MacOS Catalina

https://techsviewer.com/install-macos-10-14-mojave-virtualbox-windows/

DigiPAKISTAN
Towards Digital Pakistan

Hackintosh Installation for MacOS

https://www.olarila.com/forum/20-guides-and- tutorials/

https://hackintosh.com

WHAT IS A DATA TYPE?

- **A data type is the type of data (value) a variable or constant can store in it. A variable's data type determines the values that the variable can contain and the operations that can be performed on it.**

- **Another important part of a data type is its size. This specifies the size of data that can be stored in a given variable or constant.**

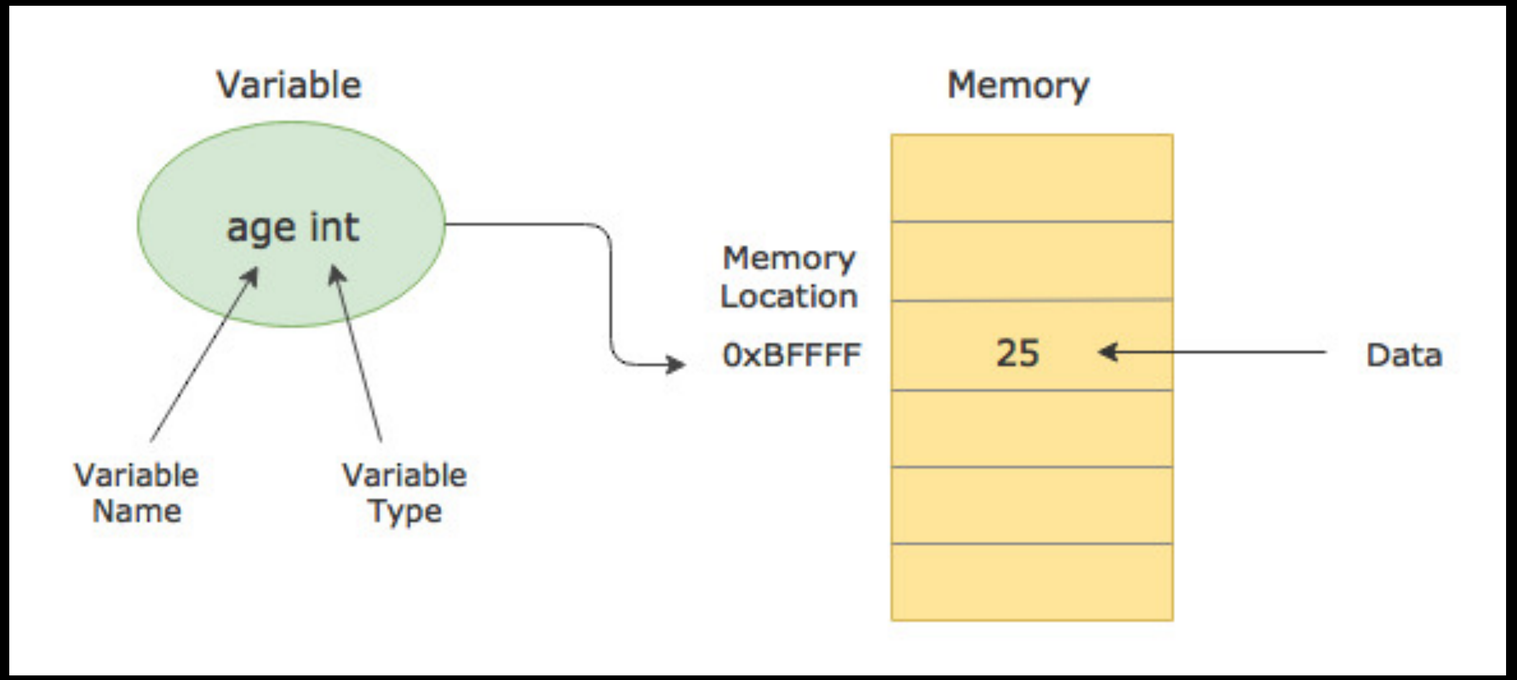
Data Type	Description	Example
Integer	**Int** or **UInt** stand for Integer and used for numbers like 1, 2, 3, 4, 5. We can use Int32, Int64 to define 32 or 64-bit signed integer, whereas UInt32 or UInt64 to define 32 or 64-bit unsigned integer variables. For example, 345566 and -345566.	100
Float	**Float** stands for floating value and it is used to represent numbers with fraction values like 245.344 and etc. It is used to hold the numbers with larger or smaller decimal points like 3.14, and -455.3344.	3.14 -455.3344
Double	**Double** stands for Double means if we want to allocate the large number with fraction value then we use Double as a float data-type. We can use it as a 64-bit floating-point number and Double is more precise than Float. For example, 345.344544, and -4554.3455543.	345.344544 -4554.3455543

Bool	**Bool** represents only two values either TRUE or FALSE. We can use Bool to check whether certain conditions met or not.	let i = 10
		if i> 20 {
		// your code
		}

String	**String** is a combination of Characters. We can define a string data type by adding double quotes to our text like "Hello World".	"" "welcome to tutlane"
Character	It represents a single alphabet character like "H", "e", "L", "L", "o".	"H"

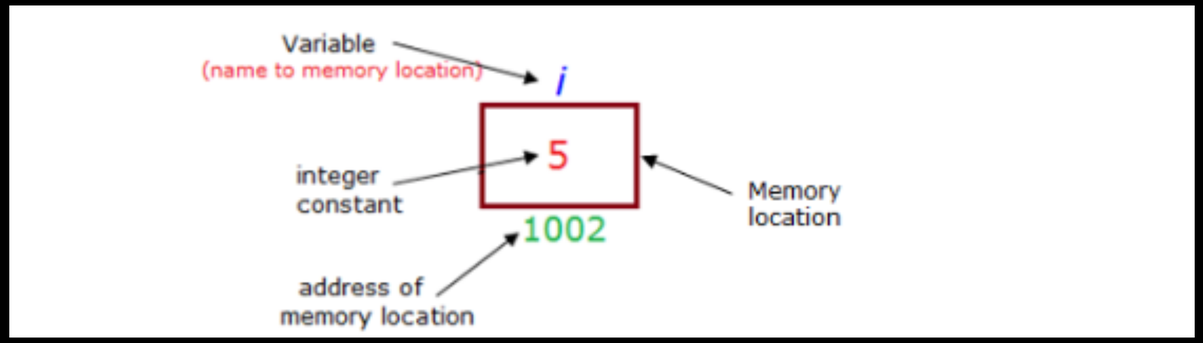
VARIABLES

VARIABLES: **Variables are the names you give to computer memory locations which are used to store values in a computer program**



CONSTANTS

CONSTANT: **Constants refer to fixed values that a program may not alter during its execution. Constants can be of any of the basic data types like an integer constant, a floating constant, a character constant, or a string literal.**



DigiPAKISTAN
Towards Digital Pakistan

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

VAR

var is a keyword which is used to declare **mutable/changeable** variables in swift

The *syntax* for declaring and initializing, as shown in the previous example, is as follows:

1. `var` is the *keyword* that indicates a new variable declaration
2. `age` is the *name* of the variable
3. `:` separates the variable *name* and *type*
4. `Int` is the *type* of the variable
5. `=` is used to *assign* a *value* to the variable
6. `42` is the *value* of the variable

Can you also change a variable's value? Yes! Like this:

```
1 var age:Int = 42
2 age = 999
3 print(age)
```

LET

let is a keyword which is used to declare **immutable/unchangeable** variables in swift

In Swift you can create *variables* with `var` and *constants* with `let`. The difference between a variable and a constant is that a variable can be changed once it's set, and a constant cannot.

You declare a *constant* with the `let` keyword, like this:

```
1  let name:String = "Bob"
2  print(name)
```

PLAY

Hide warnings

See how the syntax is exactly the same as before, except for the `let` keyword? And keep in mind that you can't change a *constant* after it has been initialized.

This code will result in an error:

```
1  let name:String = "Bob"
2  name = "Alice"
3  print(name)
```

EXPLICIT VS IMPLICIT DECLARATION

DigiPAKISTAN
Towards Digital Pakistan

EXPLICIT DECLARATION:

Explicit is the manual approach to accomplishing the change you wish to have by writing out the instructions to be done explicitly

```
var age: Int = 27
let name: String = "iOS Training"
```

IMPLICIT DECLARATION:

Implicit is often used to refer to something that’s done for you by other code behind the scenes.

```
var age = 27
let name = "iOS Training"
```

EXPRESSIONS, VARIABLES & CONSTANTS

Constants are useful for values that aren't going to change its values

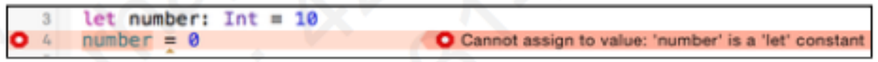
Once you've declared a constant, you can't change its data. For example, consider the following code:

```
let number: Int = 10
number = 0
```

This code produces an error:

```
Cannot assign to value: 'number' is a 'let' constant
```

In Xcode, you would see the error represented this way:



When you will need to change some data, you should use a variable to represent that data instead of a constant.You declare a v way

```
var variableNumber: Int = 42
```

Only the first part of the statement is different: You declare constants using `let`, whereas you declare variables using `var`.

Once you've declared a variable, you're free to change it to whatever you wish, as long as the type remains the same. For example, to change the variable declared above, you could do this:

```
var variableNumber: Int = 42
variableNumber = 0
variableNumber = 1_000_000
```

To change a variable, you simply assign it a new value.

Note: In Swift, you can optionally use underscores to make larger numbers more human-readable. The quantity and placement of the underscores is up to you.

PRACTICE QUESTIONS

Declare a constant of type `Int` called `myAge` and set it to your age.

Declare a variable of type `Double` called `averageAge`. Initially, set it to your own age. Then, set it to the average of your age and my own age of `30`.

Declare a constant `Int` called `myAge` and set it equal to your age. Also declare an `Int` variable called `dogs` and set it equal to the number of dogs you own. Then imagine you bought a new puppy and increment the `dogs` variable by one.

Given the following code:

```
age: Int = 16
print(age)
age = 30
print(age)
```

Modify the first line so that it compiles. Did you use `var` or `let`?

The resistance of such an appliance can be then calculated (in a long-winded way) as the power divided by the current squared. Calculate the resistance and store it in a constant called `resistance` of type `Double`.

PRACTICE QUESTIONS

Consider the following code:

```
let x: Int = 46
let y: Int = 10
```

Work out what `answer` equals when you add the following lines of code:

```
// 1
let answer1: Int = (x * 100) + y
// 2
let answer2: Int = (x * 100) + (y * 100)
// 3
let answer3: Int = (x * 100) + (y / 10)
```

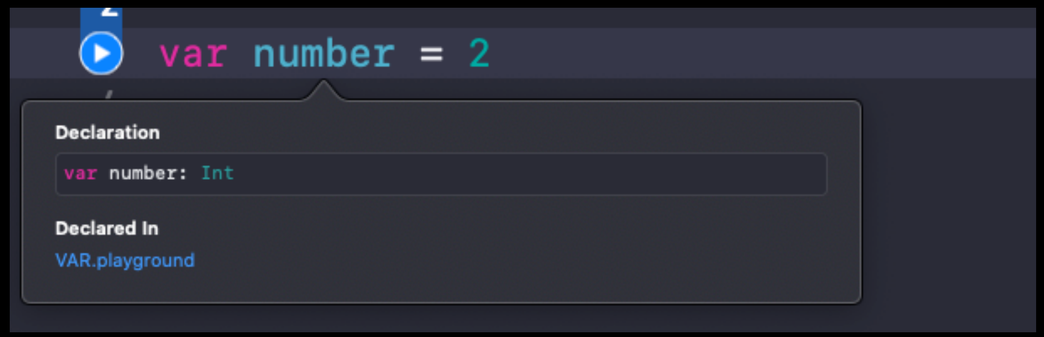
Add as many parentheses to the following calculation, ensuring that it doesn't change the result of the calculation.

```
8 - 4 * 2 + 6 / 3 * 4
```

Declare three constants called `rating1`, `rating2` and `rating3` of type `Double` and assign each a value. Calculate the average of the three and store the result in a constant named `averageRating`.

The power of an electrical appliance can be calculated by multiplying the voltage by the current. Declare a constant named `voltage` of type `Double` and assign it a value. Then declare a constant called `current` of type `Double` and assign it a value. Finally calculate the power of the electrical appliance you've just created storing it in a constant called `power` of type `Double`.

WHAT IS TYPE INFERENCE?



- If you don't specify the **type** of value you need, **Swift** uses **type inference** to work out the appropriate **type**.
- **Type inference** enables a compiler to deduce the **type** of a particular expression automatically when it compiles your code, simply by examining the values you provide.

WHAT IS TYPE CONVERSION?

TYPE CONVERSION IS THE PROCESS IN WHICH WE CAN CONVERT THE TYPE OF ONE OBJECT INTO ANOTHER TYPE

Remember, computers rely on us programmers to tell them what to do. In Swift, that includes being explicit about type conversions. If you want the conversion to happen, you have to say so!

Instead of simply assigning, you need to explicitly say that you want to convert the type. You do it like so:

```
integer = Int(decimal)
```

The assignment on the third line now tells Swift unequivocally that you want to convert from the original type, `Double`, to the new type, `Int`.

> **Note**: In this case, assigning the decimal value to the integer results in a loss of precision: The `integer` variable ends up with the value `12` instead of `12.5`. This is why it's important to be explicit. Swift wants to make sure you know what you're doing and that you may end up losing data by performing the type conversion.

TYPES AND OPERATIONS

Type Conversion:

Sometimes you'll have data in one format and need to convert it to another. The naïve way to attempt this would be like so:

```
var integer: Int = 100
var decimal: Double = 12.5
integer = decimal
```

Swift will complain if you try to do this and spit out an error on the third line:

```
Cannot assign value of type 'Double' to type 'Int'
```

Some programming languages aren't as strict and will perform conversions like this automatically. Experience shows this kind of automatic conversion is the source of software bugs and often hurts performance. Swift disallows you from assigning a value of one type to another and avoids these issues.

Swift Performs explicit type conversions

Instead of simply assigning, you need to explicitly say that you want to convert the type. You do it like so:

```
var integer: Int = 100
var decimal: Double = 12.5
integer = Int(decimal)
```

The assignment on the third line now tells Swift unequivocally that you want to convert from the original type, `Double`, to the new type, `Int`.

> **Note**: In this case, assigning the decimal value to the integer results in a loss of precision: The `integer` variable ends up with the value `12` instead of `12.5`. This is why it's important to be explicit. Swift wants to make sure you know what you're doing and that you may end up losing data by performing the type conversion.

TYPES AND OPERATIONS

Type Conversions

You might think you could do it like this:

```
let hourlyRate: Double = 19.5
let hoursWorked: Int = 10
let totalCost: Double = hourlyRate * hoursWorked
```

If you try that, you'll get an error on the final line:

```
Binary operator '*' cannot be applied to operands of type 'Double' and
'Int'
```

This is because in Swift, you can't apply the * operator to mixed types. This rule also applies to the other arithmetic operators. It may seem surprising at first, but Swift is being rather helpful.

You need to tell Swift you want it to consider the `hoursWorked` constant to be a `Double`, like so:

```
let hourlyRate: Double = 19.5
let hoursWorked: Int = 10
let totalCost: Double = hourlyRate * Double(hoursWorked)
```

Now, each of the operands will be a `Double` when Swift multiplies them, so `totalCost` is a `Double` as well.

TYPES AND OPERATIONS

Type Inference:

- Type Inference used in swift to automatically detect a data type of a certain variable if its data type is not explicitly declared by a user.

Sometimes you want to define a constant or variable and ensure it's a certain type, even though what you're assigning to it is a different type. You saw earlier how you can convert from one type to another. For example, consider the following:

```
let wantADouble = 3
```

Here, Swift infers the type of `wantADouble` as `Int`. But what if you wanted `Double` instead?

The first thing you could do is the following:

```
let actuallyDouble = Double(3)
```

This is like you saw before with type conversion.

Another option would be to not use type inference at all and do the following:

```
let actuallyDouble: Double = 3
```

There is a third option, like so:

```
let actuallyDouble = 3 as Double
```

PRACTICE QUESTIONS

1. Create a constant called `age1` and set it equal to `42`. Create a constant called `age2` and set it equal to `21`. Check using Option-click that the type for both has been inferred correctly as `Int`.
2. Create a constant called `avg1` and set it equal to the average of `age1` and `age2` using the naïve operation `(age1 + age2) / 2`. Use Option-click to check the type and check the result of `avg1`. Why is it wrong?
3. Correct the mistake in the above exercise by converting `age1` and `age2` to type `Double` in the formula. Use Option-click to check the type and check the result of `avg1`. Why is it now correct?

TYPES & OPERATIONS

DigiPAKISTAN
Towards Digital Pakistan

Strings:

Swift, like any good programming language, can work directly with characters and strings. It does so through the data types `Character` and `String`, respectively. In this section, you'll learn about these data types and how to work with them.

Characters and strings

The `Character` data type can store a single character. For example:

```
let characterA: Character = "a"
```

This stores the character **a**. It can hold any character — even an emoji:

```
let characterDog: Character = "🐶"
```

But this data type is designed to hold only single characters. The `String` data type, on the other hand, stores multiple characters. For example:

```
let stringDog: String = "Dog"
```

It's as simple as that! The right-hand side of this expression is what's known as a **string literal**; it's the Swift syntax for representing a string.

Of course, type inference applies here as well. If you remove the type in the above declaration, then Swift does the right thing and makes the `stringDog` a `String` constant:

```
let stringDog = "Dog" // Inferred to be of type String
```

> **Note:** There's no such thing as a character literal in Swift. A character is simply a string of length one. However, Swift infers the type of any string literal to be `String`, so if you want a `Character` instead, you must make the type explicit.

TYPES & OPERATIONS

Concatenation and Interpolation in Strings:

you can add numbers, you can add strings:

```
var message = "Hello" + " my name is "
let name = "Matt"
message += name // "Hello my name is Matt"
```

You need to declare `message` as a variable rather than a constant because you want to modify it. You can add string literals together, as in the first line, and you can add string variables or constants together, as in the last line.

It's also possible to add characters to a string. However, Swift's strictness with types means you have to be explicit when doing so, just as you have to be when you work with numbers if one is an `Int` and the other is a `Double`.

To add a character to a string, you do this:

```
let exclamationMark: Character = "!"
message += String(exclamationMark) // "Hello my name is Matt!"
```

With this code, you explicitly convert the `Character` to a `String` before you add it to `message`.

Interpolation

You can also build up a string by using **interpolation**, which is a special Swift syntax that lets you build a string in a way that's easy to read:

```
let name = "Matt"
let message = "Hello my name is \(name)!" // "Hello my name is Matt!"
```

As I'm sure you'll agree, this is much more readable than the example from the previous section. It's an extension of the string literal syntax, whereby you replace certain parts of the string with other values. You enclose the value you want to give the string in parentheses preceded by a backslash.

This syntax works in just the same way to build a string from other data types, such as numbers:

```
let oneThird = 1.0 / 3.0
let oneThirdLongString = "One third is \(oneThird) as a decimal."
```

Here, you use a `Double` in the interpolation. At the end of this code, your `oneThirdLongString` constant will contain the following:

```
One third is 0.3333333333333333 as a decimal.
```

TYPES & OPERATION

DigiPAKISTAN
Towards Digital Pakistan

Mini Exercises:

1. Create a string constant called `firstName` and initialize it to your first name. Also create a string constant called `lastName` and initialize it to your last name.
2. Create a string constant called `fullName` by adding the `firstName` and `lastName` constants together, separated by a space.
3. Using interpolation, create a string constant called `myDetails` that uses the `fullName` constant to create a string introducing yourself. For example, my string would read: `"Hello, my name is Matt Galloway."`.

WHAT ARE TUPLES?

```
let profileInfo: (name: String, id: Int) = ("Farhaj Ahmed", 104)
```

- **Tuples** are used to group multiple values together into single compound value. The values whatever we will define in a tuple can be of any type and there is no restriction to use same type of values.
- In **Swift** we have two types of tuples are available those are **unnamed** tuples and **named** tuples.

TYPES & OPERATIONS

TUPLES

Sometimes data comes in pairs or triplets. An example of this is a pair of (x, y) coordinates on a 2D grid. Similarly, a set of coordinates on a 3D grid is comprised of an x-value, a y-value and a z-value.

In Swift, you can represent such related data in a very simple way through the use of a *tuple*.

A tuple is a type that represents data composed of more than one value of any type. You can have as many values in your tuple as you like. For example, you can define a pair of 2D coordinates where each axis value is an integer, like so:

```
let coordinates: (Int, Int) = (2, 3)
```

The type of `coordinates` is a tuple containing two `Int` values. The types of the values within the tuple, in this case `Int`, are separated by commas surrounded by parentheses. The code for creating the tuple is much the same, with each value separated by commas and surrounded by parentheses.

Type inference can infer tuple types too:

```
let coordinates = (2, 3)
```

You could similarly create a tuple of `Double` values, like so:

```
let coordinates = (2.1, 3.5)
// Inferred to be of type (Double, Double)
```

Or you could mix and match the types comprising the tuple, like so:

```
let coordinates = (2.1, 3)
// Inferred to be of type (Double, Int)
```

TYPES & OPERATIONS

TUPLES

And here's how to access the data inside a tuple:

```
let coordinates = (2, 3)
let x = coordinates.0
let y = coordinates.1
```

You can reference each item in the tuple by its position in the tuple, starting with zero. So in this example, `x` will equal `2` and `y` will equal `3`.

In the previous example, it may not be immediately obvious that the first value, at index `0`, is the x-coordinate and the second value, at index `1`, is the y-coordinate. This is another demonstration of why it's important to *always* name your variables in a way that avoids confusion.

Fortunately, Swift allows you to name the individual parts of a tuple and you can be explicit about what each part represents. For example:

```
let coordinatesNamed = (x: 2, y: 3)
// Inferred to be of type (x: Int, y: Int)
```

Here, the code annotates the values of `coordinatesNamed` to contain a label for each part of the tuple.

Then, when you need to access each part of the tuple, you can access it by its name:

```
let x = coordinatesNamed.x
let y = coordinatesNamed.y
```

This is much clearer and easier to understand. More often than not, it's helpful to name the components of your tuples.

If you want to access multiple parts of the tuple at the same time, as in the examples above, you can also use a shorthand syntax to make it easier:

```
let coordinates3D = (x: 2, y: 3, z: 1)
let (x, y, z) = coordinates3D
```

This declares three new constants, `x`, `y` and `z`, and assigns each part of the tuple to them in turn. The code is equivalent to the following:

```
let coordinates3D = (x: 2, y: 3, z: 1)
let x = coordinates3D.x
let y = coordinates3D.y
let z = coordinates3D.z
```

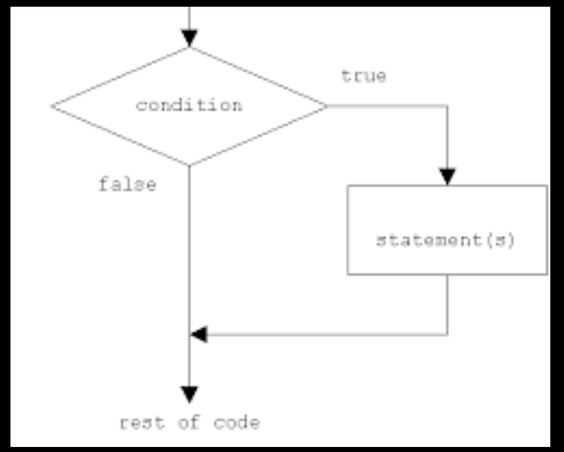
If you want to ignore a certain element of the tuple, you can replace the corresponding part of the declaration with an underscore. For example, if you were performing a 2D calculation and wanted to ignore the z-coordinate of `coordinates3D`, then you'd write the following:

```
let (x, y, _) = coordinates3D
```

This line of code only declares `x` and `y`. The `_` is special and simply means you're ignoring this part for now.

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

BASIC CONTROL FLOW

This is how you use a Boolean in Swift:

```
let yes: Bool = true
let no: Bool = false
```

And because of Swift's type inference, you can leave off the type:

```
let yes = true
let no = false
```

Booleans are commonly used to compare values. For example, you may have two values and you want to know if they're equal: either they are (true) or they aren't (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)
```

Because 1 does not equal 2, (1==2) is `false`, and then `!` flips it 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)
```

BASIC CONTROL FLOW

```
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
```

The parentheses disambiguates the expression. First Swift evaluates the sub-expression inside the parentheses, and then it evaluates the full expression, following these steps:

```
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"
```

Here, `dogEqualsCat` is a Boolean that in this case equals `false`, because `"dog"` does not equal `"cat"`. Simple!

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"`.

BASIC CONTROL FLOW

```
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
```

The parentheses disambiguates the expression. First Swift evaluates the sub-expression inside the parentheses, and then it evaluates the full expression, following these steps:

```
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"
```

Here, `dogEqualsCat` is a Boolean that in this case equals `false`, because `"dog"` does not equal `"cat"`. Simple!

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"`.

BASIC CONTROL FLOW

If else Statement in Swift

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

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)
```

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

The Ternary Conditional Operator:

The tertiary conditional operator takes a condition and return 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`.

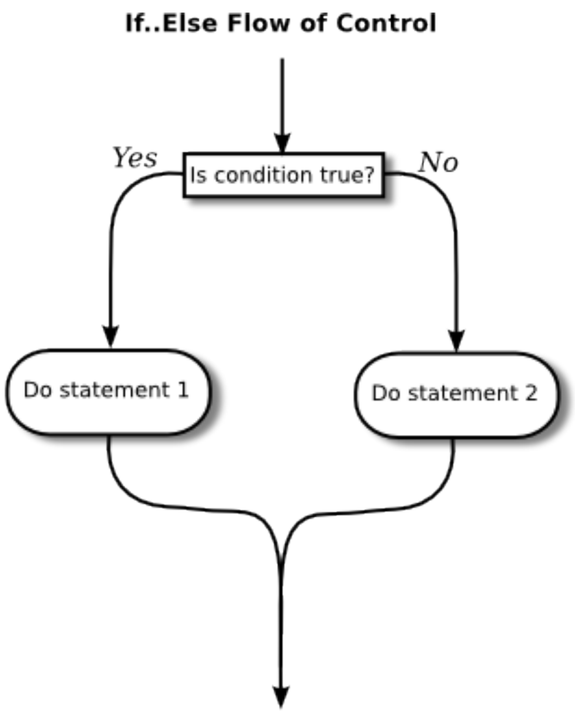
I'm sure you agree that's much simpler! This is a useful operator that you'll find yourself using regularly.

> **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.

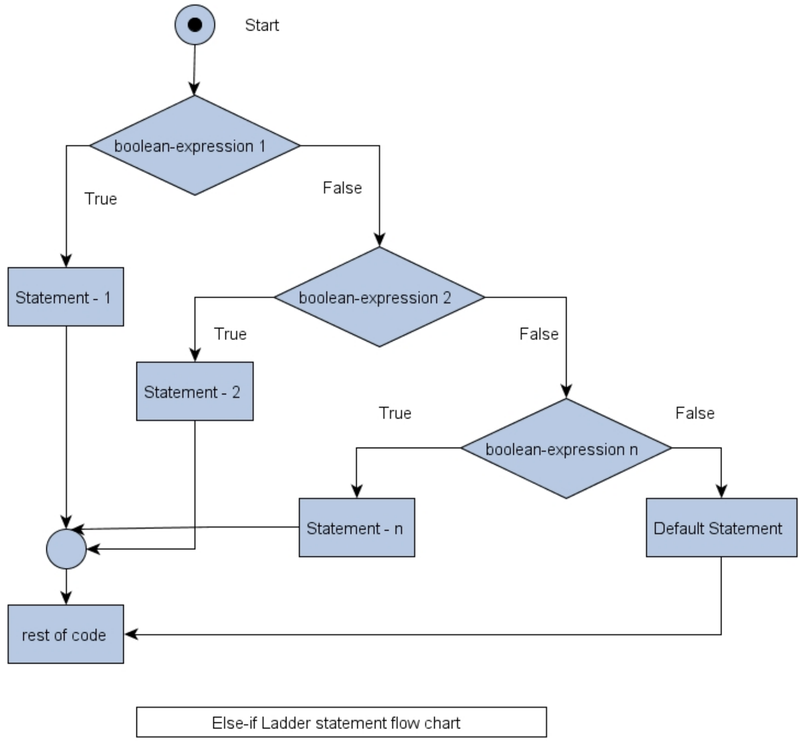
DigiPAKISTAN
Towards Digital Pakistan

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.

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`.

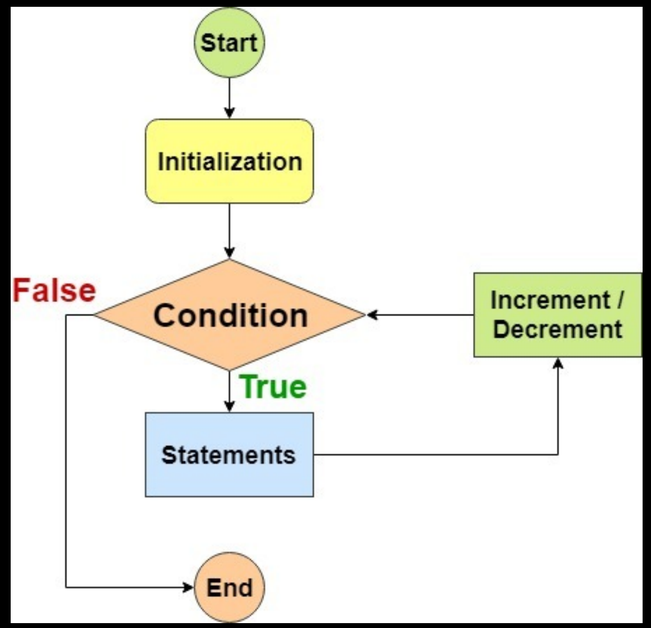
What's wrong with the following code?

```
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

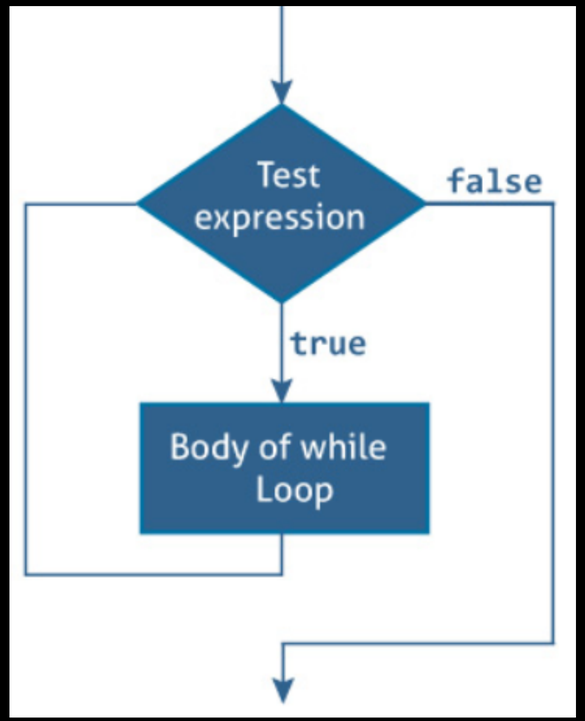
- WHILE LOOP
- REPEAT WHILE LOOP
- FOR LOOP

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.



DigiPAKISTAN
Towards Digital Pakistan

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

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
}
```

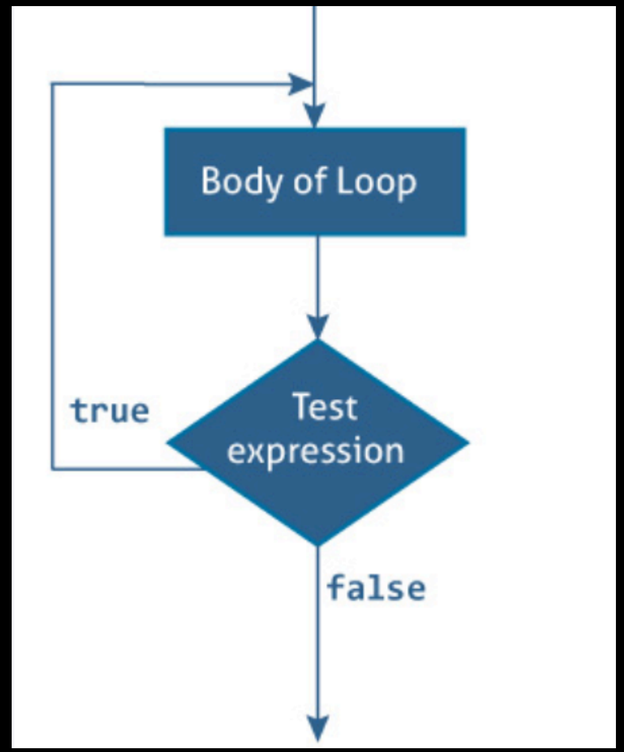
This process goes on until the TestExpression is evaluated to false. If the TestExpression evaluates to false, while loop is terminated.

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)
```

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.

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

DigiPAKISTAN
Towards Digital Pakistan

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

RULES FOR TUPLES

Tuples allow you to store several values together in a single value. That might sound like arrays, but tuples are different:

1. You can't add or remove items from a tuple; they are fixed in size.

2. You can't change the type of items in a tuple; they always have the same types they were created with.

3. You can access items in a tuple using **numerical** positions or by **naming** them, but Swift won't let you read numbers or names that don't exist.

WHAT IS THE PURPOSE OF _ IN TUPLES?

```
let http200Success = (200, "Success")

let (responseCode, responseMessage) = http200Success

let (justTheResponseCode, _) = http200Success
```

You'll find that you can use the underscore (also called the wildcard operator) throughout Swift to ignore a value.

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

PRACTICE QUESTIONS

What is the type of the constant named `value`?

```
let tuple = (100, 1.5, 10)
let value = tuple.1
```

What is the value of the constant named `month`?

```
let tuple = (day: 15, month: 8, year: 2015)
let month = tuple.month
```

PRACTICE QUESTIONS

1. Declare a constant tuple that contains three `Int` values followed by a `Double`. Use this to represent a date (month, day, year) followed by an average temperature for that date.
2. Change the tuple to name the constituent components. Give them names related to the data that they contain: `month`, `day`, `year` and `averageTemperature`.
3. In one line, read the day and average temperature values into two constants. You'll need to employ the underscore to ignore the month and year.
4. Up until now, you've only seen constant tuples. But you can create variable tuples, too. Change the tuple you created in the exercises above to a variable by using `var` instead of `let`. Now change the average temperature to a new value.

DigiPAKISTAN
Towards Digital Pakistan

WHAT ARE BOOLEANS?

```
let isFileExist: Bool = false
```

- A **Bool** **is** value type whose instances are either **true** or **false**.
- **Bool** represents **Boolean** values in Swift. Create instances of Bool by using one of the Boolean literals **true** or **false**, or by assigning the result of a Boolean method or operation to a **variable** or **constant**.

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)
```

DigiPAKISTAN
Towards Digital Pakistan

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

PRACTICE QUESTIONS

1. Create a constant called `myAge` and set it to your age. Then, create a constant named `isTeenager` that uses Boolean logic to determine if the age denotes someone in the age range of 13 to 19.
2. Create another constant named `theirAge` and set it to my age, which is 30. Then, create a constant named `bothTeenagers` that uses Boolean logic to determine if both you and I are teenagers.
3. Create a constant named `reader` and set it to your name as a string. Create a constant named `author` and set it to my name, Matt Galloway. Create a constant named `authorIsReader` that uses string equality to determine if `reader` and `author` are equal.
4. Create a constant named `readerBeforeAuthor` which uses string comparison to determine if `reader` comes before `author`.

In each of the following statements, what is the value of the Boolean `answer` constant?

```
let answer = true && true
let answer = false || false
let answer = (true && 1 != 2) || (4 > 3 && 100 < 1)
let answer = ((10 / 2) > 3) && ((10 % 2) == 0)
```

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