iOS Development with Swift

Xcode & Swift Basics

Week 2

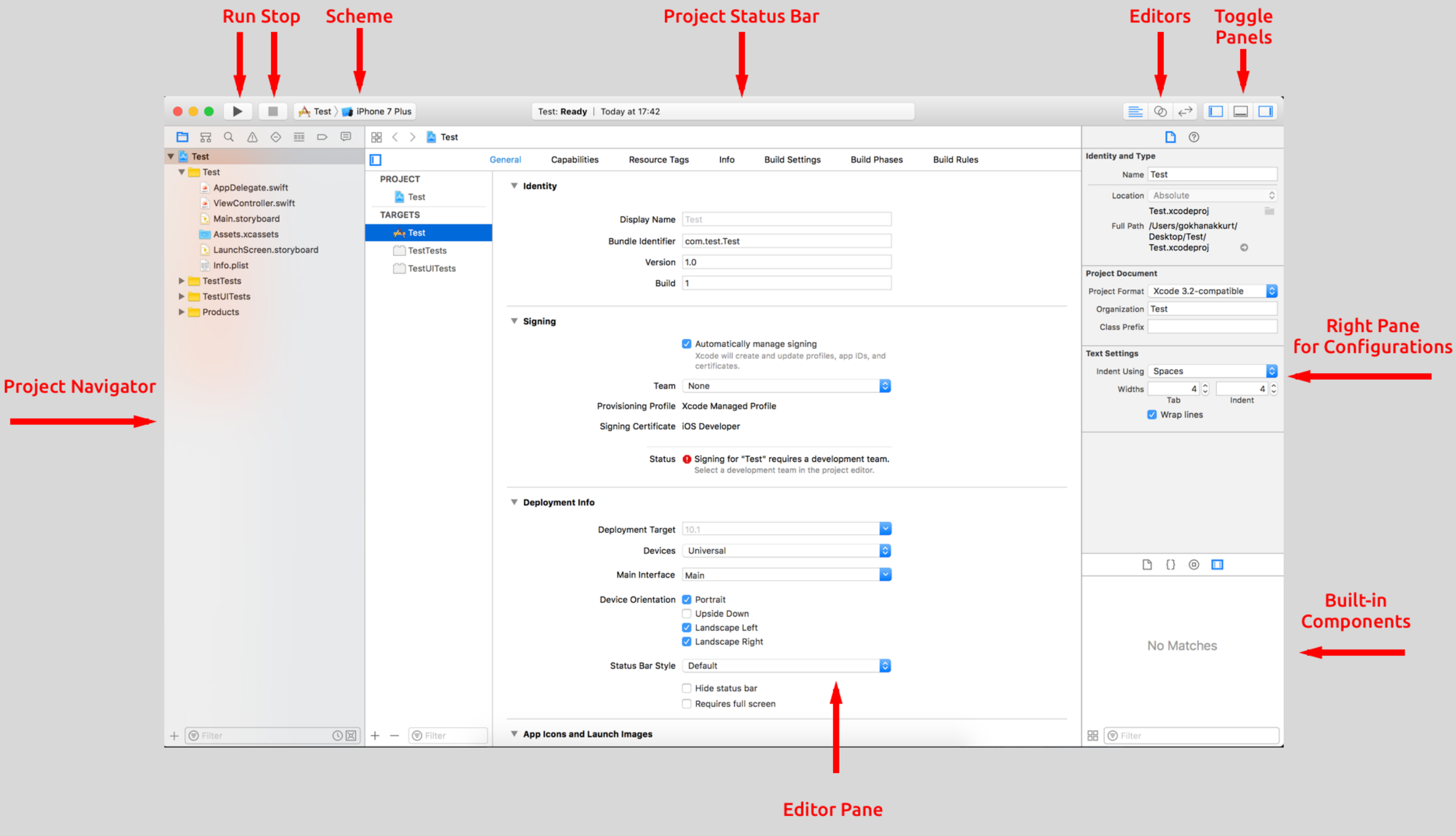
Outline

- Glance at Xcode
- Introduction Swift
- Swift Basics
- Operators
- Strings and Characters
- Collection Types
- Control Flows
- Workshop

Xcode


Welcome to Xcode
Version 8.1 (8B62)
Get started with a playground
Explore new ideas quickly and easily.
Create a new Xcode project
Create an app for iPhone, iPad, Mac, Apple Watch or Apple TV.
Check out an existing project
Start working on something from an SCM repository.
Powerbar
~/Developer/ios/barcin-powerbar
SideMenuController
~/Downloads/SideMenuController-master
Pods
.../ios/barcin-datepicker/DatePicker/Example
DatePicker
.../ios/barcin-datepicker/DatePicker/Example
BarcinCard
~/Developer/ios/barcin-card
BarcinStats
~/Developer/ios/barcin-stats
Open another project...

Xcode


Run Stop
Scheme
Project Status Bar
Editors
Toggle Panels
Test 〉 iPhone 7 Plus
Test: Ready | Today at 17:42
Test
General
Capabilities
Resource Tags
Info
Build Settings
Build Phases
Build Rules
PROJECT
Test
TARGETS
Test
TestTests
TestUITests
Identity
Display Name Test
Bundle Identifier com.test.Test
Version 1.0
Build 1
Signing
Automatically manage signing
Xcode will create and update profiles, app IDs, and certificates.
Team None
Provisioning Profile Xcode Managed Profile
Signing Certificate iOS Developer
Status Signing for "Test" requires a development team.
Select a development team in the project editor.
Deployment Info
Deployment Target 10.1
Devices Universal
Main Interface Main
Device Orientation Portrait
Upside Down
Landscape Left
Landscape Right
Status Bar Style Default
Hide status bar
Requires full screen
App Icons and Launch Images
Identity and Type
Name Test
Location Absolute
Test.xcodeproj
Full Path /Users/gokhanakkurt/Desktop/Test/Test.xcodeproj
Project Document
Project Format Xcode 3.2-compatible
Organization Test
Class Prefix
Text Settings
Indent Using Spaces
Widths 4 4
Tab Indent
Wrap lines
No Matches
Filter
Project Navigator
Right Pane for Configurations
Built-in Components
Editor Pane

Introduction to Swift

- Swift is a new programming language for iOS, macOS, watchOS, and tvOS app development.
- For students, learning Swift has been a great introduction to modern programming concepts and best practices.
- Many parts of Swift will be familiar from your experience of developing in C and Objective-C

Introduction to Swift

- Swift has its own built-in data types like other programming languages C, C++ and Objective-C
- These are **Int** for integers, **Float** and **Double** floating-point values, **Bool** for boolean values, **String** for textual data.
- **Array**, **Set** and **Dictionary** are three main collection types in Swift.

Constants and Variables

- Constants and variables associate a name with a value of a particular type.
- The value of a ***constant*** cannot be changed once it is set, whereas a ***variable*** can be set to a different value in the future.

```
let maximumNumberOfLoginAttempts = 10
var currentLoginAttempt = 0
```

Type Annotations

You can provide a type annotation when you declare a constant or variable, to be clear about the kind of values the constant or variable can store.

```
let helloWorld : String = "Hello World!"

let pi : Float = 3.14159

let currentYear : Int = 2016
```

Naming Constants and Variables

Constant and variable names can contain almost any character, including Unicode characters:

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Printing Constants and Variables

- You can print the current value of a constant or variable with **print(_:separator:terminator:)** function

```
let helloWorld : String = "Hello World!"
print(helloWorld)
```

- You can either pass the parameter as following

```
let helloWorld : String = "Hello World!"
print("It says : \(helloWorld)")
```

Comments

Use comments to include nonexecutable text in your code, as a note or reminder to yourself.

```
// This is a comment.
```

```
/* This is the start of the first multiline comment.
 /* This is the second, nested multiline comment. */
 This is the end of the first multiline comment. */
```

Semicolons

- Unlike many other languages, Swift does not require you to write a semicolon **(;)** after each statement in your code.
- However, semicolons are required if you want to write multiple separate statements on a single line:

```
let cat = "🐱"; print(cat)
// Prints "🐱"
```

Integers

- Integers are whole numbers with no fractional component, such as 23 and -42. Stated as **Int**
- Integers are either signed (positive, zero, or negative) or unsigned (positive or zero).
- Swift provides signed and unsigned integers in 8, 16, 32, and 64 bit forms.
- These are **UInt8**, **UInt16**, **UInt32**, **UInt64**

Floating-Point Numbers

- Floating-point numbers are numbers with a fractional component, such as 3.14159, 0.1, and -273.15.
- **Double** represents a 64-bit floating-point number.
- **Float** represents a 32-bit floating-point number.

Booelan

Swift has a basic **Boolean** type, called **Bool**. Boolean values are referred to as logical, because they can only ever be true or false. Swift provides two Boolean constant values, **true** and **false**:

```
let orangesAreOrange = true
let turnipsAreDelicious = false
```

Type Safety and Inference

Swift is a type-safe language. A type safe language encourages you to be clear about the types of values your code can work with. If part of your code expects a String, you can't pass it an Int by mistake.

```
let meaningOfLife = 42
// meaningOfLife is inferred to be of type Int
```

```
let pi = 3.14159
// pi is inferred to be of type Double
```

Basic Operators

- An operator is a special symbol or phrase that you use to check, change, or combine values.
- For instance, **(+)** operator adds two numbers.
- Unary operators operate on a single target (such as -a).
- Binary operators operate on two targets (such as 2 + 3)

```
let x = 2 + 3
let y = 3 - 5
let z = 10 / 2
let w = 20 * 3
```

Assignment Operator

The assignment operator (a = b) initializes or updates the value of a with the value of b:

```
let b = 10
var a = 5
a = b
// a is now equal to 10
```

Arithmetic Operators

Swift supports the four standard arithmetic operators for all number types:

- Addition **(+)**
- Subtraction **(-)**
- Multiplication **(*)**
- Division **(/)**

```
1 + 2       // equals 3
5 - 3       // equals 2
2 * 3       // equals 6
10.0 / 2.5  // equals 4.0
```

Remainder Operator

The remainder operator (a % b) works out how many multiples of b will fit inside a and returns the value that is left over (known as the remainder).

```
9 % 4    // equals 1
```

Compound Assignment Operators

Like C, Swift provides compound assignment operators that combine assignment (=) with another operation. One example is the addition assignment operator (+=)

```
var a = 1
a += 2 // a = a + 2
// a is now equal to 3
```

What about this?

```
var a = 3
a -= 20
print(a)
```

Comparison Operators

Swift supports all standard C comparison operators:

- Equal to **(a == b)**
- Not equal to **(a != b)**
- Greater than **(a > b)**
- Less than **(a < b)**
- Greater than or equal to **(a >= b)**
- Less than or equal to **(a <= b)**

An Example

```
1 == 1   // true because 1 is equal to 1
2 != 1   // true because 2 is not equal to 1
2 > 1    // true because 2 is greater than 1
1 < 2    // true because 1 is less than 2
1 >= 1   // true because 1 is greater than or equal to 1
2 <= 1   // false because 2 is not less than or equal to 1
```

Logical Operators

Logical operators modify or combine the Boolean logic values true and false. Swift supports the three standard logical operators found in C-based languages:

- Logical **NOT** (!a)
- Logical **AND** (a && b)
- Logical **OR** (a || b)

NOT Operator

The logical NOT operator (!a) inverts a Boolean value so that true becomes false, and false becomes true.

```
let allowedEntry = false
if !allowedEntry {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

AND Operator

The logical AND operator (a && b) creates logical expressions where both values must be true for the overall expression to also be true.

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

OR Operator

The logical OR operator (a || b) is used it to create logical expressions in which only one of the two values has to be true for the overall expression to be true.

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

Strings and Characters

A string is a series of characters, such as "hello, world" or "albatross". Swift strings are represented by the String type. The contents of a String can be accessed in various ways, including as a collection of Character values.

```
let helloWorld : String = "Hello World!"
print(helloWorld.characters.count)
// prints 12
```

Initialize String

To create an empty String value as the starting point for building a longer string, either assign an empty string literal to a variable, or initialize a new String instance with initializer syntax:

```
var emptyString = ""                // empty string literal
var anotherEmptyString = String()  // initializer syntax
// these two strings are both empty, and are equivalent to each other
```

Initialize String Cont'd

Find out whether a String value is empty by checking its Boolean **isEmpty** property:

```
if emptyString.isEmpty {
    print("Nothing to see here")
}
// Prints "Nothing to see here"
```

String Mutability

You indicate whether a particular String can be modified (or mutated) by assigning it to a variable, or to a constant (in which case it cannot be modified):

```
var variableString = "Horse"
variableString += " and carriage"
// variableString is now "Horse and carriage"

let constantString = "Highlander"
constantString += " and another Highlander"
// this reports a compile-time error - a constant string cannot be modified
```

Characters

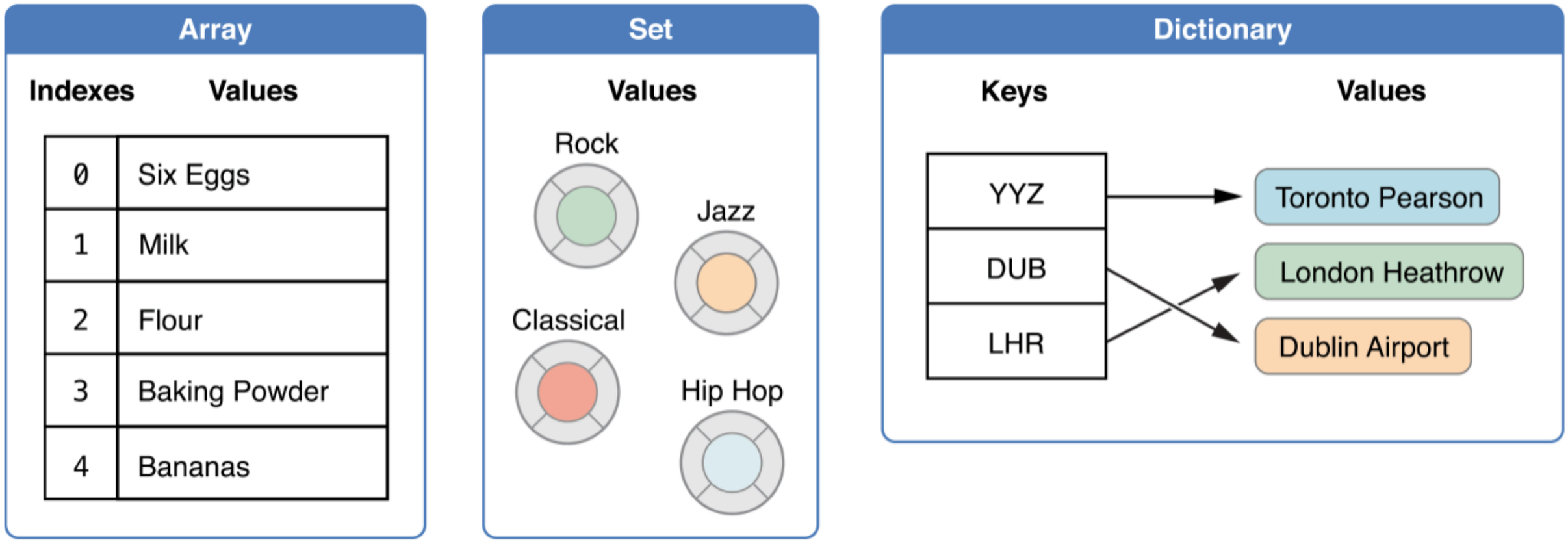
You can access the individual Character values for a String by iterating over its characters property

```
for character in "Dog!🐶".characters {
    print(character)
}
// D
// o
// g
// !
// 🐶
```

Collection Types

Swift provides three primary collection types, known as **arrays**, **sets**, and **dictionaries**, for storing collections of values. **Arrays** are ordered collections of values. **Sets** are unordered collections of unique values. **Dictionaries** are unordered collections of key-value associations.

Array, Set and Dictionary



Arrays

An array stores values of the same type in an ordered list. The same value can appear in an array multiple times at different positions.

```
let values : [Int] = [3, 2, 1, 10, 21, 24]
```

```
let fruits : [String] = [
    "strawberry",
    "melon",
    "banana",
    "grape",
    "apricot"
]
```

Sets

A set stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items is not important, or when you need to ensure that an item only appears once.

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres has been initialized with three initial items
```

Dictionaries

A dictionary stores associations between keys of the same type and values of the same type in a collection with no defined ordering. Each value is associated with a unique key, which acts as an identifier for that value within the dictionary.

```
var airports: [String: String] = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

Control Flows

Swift provides a variety of control flow statements. These include **while** loops to perform a task multiple times; **if**, **guard**, and **switch** statements to execute different branches of code based on certain conditions; and statements such as **break** and **continue** to transfer the flow of execution to another point in your code.

For-In Loops

You use the for-in loop to iterate over a sequence, such as ranges of numbers, items in an array, or characters in a string.

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

While Loops

A while loop starts by evaluating a single condition. If the condition is true, a set of statements is repeated until the condition becomes false.

```
var index = 1
while index <= 5{
    print("\(index) times 5 is \(index * 5)")
    index += 1
}
```

Repeat While

The other variation of the while loop, known as the repeat-while loop, performs a single pass through the loop block first, before considering the loop's condition. It then continues to repeat the loop until the condition is false.

```
var x : Int = 20
repeat{
    print(x)
    x -= 1
}while x > 0
```

Conditional Statements

It is often useful to execute different pieces of code based on certain conditions. You might want to run an extra piece of code when an error occurs, or to display a message when a value becomes too high or too low.

To do this, you make parts of your code conditional.

```
let x : Int = 25
if x > 20{
    print("x is bigger than 20")
}else{
    print("x is smaller than 20")
}
```

If

In its simplest form, the if statement has a single if condition. It executes a set of statements only if that condition is true.

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// Prints "It's really warm. Don't forget to wear sunscreen."
```

Switch

A switch statement provides an alternative to the if statement for responding to multiple potential states.

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
    print("The first letter of the alphabet")
case "z":
    print("The last letter of the alphabet")
default:
    print("Some other character")
}
// Prints "The last letter of the alphabet"
```

Control Transfer Statements

Control transfer statements change the order in which your code is executed, by transferring control from one piece of code to another.

- **continue**
- break
- fallthrough
- return
- throw

```
let x : Int = 20
for i in 0...20{
    if i == x{
        continue
    }
    print(i)
}
```

```
let x : Int = 3
for i in 0...20{
    if i == x{
        break
    }
    print(i)
}
```

Workshop

Question: A is an array containing integers

A = [1, 3, 5, 7, 9, 21, 13, 10, 12]

- Print the sum of those numbers to the screen.
- Print minimum value.
- Print maximum value.
- Print “sum is bigger than 20” if sum is bigger than 20