# Unit 1: Getting started with App development

## Introduction quiz:

1. What are the three primary goals of swift?

A: safe B: accurate C: fast D: expressive

1. Which of these are features of swift that make it a safe language?

A: type safety B: type inference C: error handing D: Optionals

1. Which of the following is a great environment for prototyping swift code?

A: Xcode B: safari C: terminal D: playgrounds

1. What will the following code print to the console?

*let helloWorld = “hello, world!”*

*let helloPlayground = “hello, playground!”*

*let goodbyePlayground = “goodbye, playground!”*

*print(“Testing, testing, 1-2-3”)*

A: “hello, world” B: “hello, playground” C: “goodbye, playground” D: “Testing, testing, 1-2-3“

# Constants, variables, and data types

## Constants:

When you want to name a value that will not change during the lifetime of the program you are making.

Constants are defined by the ‘let’ keyword

let name = “John”

print (name)

you cannot give this a new value after assigning it, the code will not run.

## Variables:

When you want to name a value that may change during the lifetime of the program you will use a variable.

Variables are defined using the ‘var’ keyword

var age = 29

print (age)

because this is a variable you can add this to it and will not get an error

age = 30

print (age)

You can assign constants and variables from ither constants and variables and add and subtract them from each other.

When it comes to naming constants and variables, there are rules. Names cannot contain math symbols, spaces, and they can’t begin with a number. They should be clear and descriptive.

## Data types:

Each constant or variable has a type that describes it’s kind of value.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Purpose | Example |
| Integer | Int | Represents whole numbers | 4 |
| Double | Double | Represents number that require decimal points | 13.45 |
| Boolean | Bool | Represents true or false values | True/false |
| String | String | Represents text | “once upon a time” |

Swift also supports collection types, which group multiple values into a single constant or variable. One type is named ‘array’ which stores an ordered list of values. Another is named ‘dictionary’ which has keys that help you look up specific values.

You can also define your own types by creating a type definition.

struct Person {

let firstName: String

let lastName: String

func sayHello() {

print (“ Hi my name is \(firstName) \(lastName.”)

}

}

A type definition is basically a blueprint. They designate how to construct a building but is not a building itself and can create a lot of buildings from one type of blueprint.

A type definition declares the information it stores (properties) and its capabilities or actions (methods). In the case above a person stores String information in 2 properties and has one action (sayHello)

When you define a constant or variable you should specify a type using a type annotation or assign it a value so that the compiler can use it to infer the type.

## Constants, variables, and data types quiz:

1. Which of the following values would be best represented with a constant?

A: player name B: player level C: player score D: player location

1. Which of the following would be best represented with a variable?

A: name B: birthday C: age D: home address

1. What keyword is used to declare a constant?

A: const B: var C: let

1. Which keyword is used to declare a variable?

A: const B: var C: let

1. Which of the following is not a reason for declaring most values as constants?

A: it is a best practise B: the compiler can make special optimizations for immutable values C: it’s safer, developers can rely on a consistent value D: most values never change

1. Using type inference, which of the following variables would be assigned a Double type?

A: *var state = “Rhode Island”* B: *let country = “Belgium”* C: *let population = 142000* D: *let speedLimit = 75.0*

1. Will the following code snippet compile? Why?

*let number: Double = 3*

A: yes because the compiler will assign the value 3.0 to number B: yes because 3 is a Double C: no because 3 is not a Double

# Operators

These are the symbols that make your code work. They are used to check, change or combine values. An operator makes your code work by performing an action on the values to its left and right.

## Assign a value:

Use the = operator to assign a value. The name on the left is assigned the value on the right

let favePerson = “Luke”

The = operator is also used to modify or reassign a value

## Basic arithmetic:

You can use the +, -, \*, and / operators to perform basic maths functionality. You can use these operators to perform maths using the values of other variables. An operator can even reference the current variable and update it to a new value.

myScore = 3

myScore = myScore + 3

when you use the division operator on Int values, the result will be an Int value rounded down to the nearest whole number, unless you specifically declare the constant or variables under Double values. Make sure to use Double whenever your code requires decimal point accuracy.

let x = 51

let y = 4

let z = x/y (12)

versus

let x: Double = 51

let y: Double = 4

let z = x/y (12.75)

## Compound Assignment:

You can modify a value that’s already been assigned by using a compound assignment operator:

myScore +=3 (adds 3)

myScore -=5 (subtracts 5)

myScore \*=2 (multiplies by 2)

myScore /=2 (divides by 2)

remember that BIDMAS matters (brackets, integer, divide, multiply, add, subtract)

## Remainder Operator:

Use the remainder operator ‘%’ to quickly calculate the remainder from the division of two int values.

let dividend = 10

let divisor = 3

let quotient = dividend / divisor (3)

let remainder = dividend % divisor (1)

## Numeric Type conversion:

You can change the value of the number by prefixing the constant or variable with the type you want to convert it to.

Let x = 3

Let y = 0.1415927

Let pi = Double(x) + y

## Operators quiz:

1. What is the value of myNumber at the end of the following code?

*let x = 2*

*let y = 4*

*let z = 6*

*let myNumber = x+y+z*

A: 4 B: 6 C: 10 D: 12

1. What is the value of myNumber at the end of the following code?

*let x = 2*

*let y = 4*

*let z = 6*

*let myNumber = x+y\*z*

A: 18 B: 26 C: 32 D: 36

1. What is the value of myNumber at the end of the following code?

*let x = 2*

*let y = 4*

*let z = 6*

*let myNumber = x\*y-z*

A: 2 B: 4 C: 6 D: 12

1. What is the value of myNumber at the end of the following code?

*let x = 2*

*let y = 4*

*let z = 6*

*let myNumber = (x+y)\*z*

A: 12 B: 24 C: 36 D: 42

1. Identify the issue with the code snippet below?

*let x = 14*

*let y = 2.5*

*let result = x\*y*

A: the code will not compile. X is a Double and y is an Int B: the code will not compile. X is an Int and y is a Double C: there is no issue. Result will equal the product of x and y

1. What is the operator called when it has an arithmetic symbol in front of the equals sign =?

*myScore += 100*

A: assignment operator B: compound assignment operator C: plus equals operator D: reset operator

1. What is the value of myNumber at the end of the following code?

*Let x = 13*

*Let y = 5*

*Let myNumber = x%y*

A: 2 B: 3 C: 18 D: 65

# Control Flow

## Logical and comparison operators:

Each if statement uses a logical or comparison operator to decide if something is true or false. The result determines whether to run the block of code or skip it.

|  |  |  |
| --- | --- | --- |
| Operator | Type | Description |
| == | Comparison | Two items must be equal |
| != | Comparison | The values must not be equal to each other |
| > | Comparison | Value on the left must be greater than the value on the right |
| >= | Comparison | Value on the left must be greater than or equal to the value on the right |
| < | Comparison | Value on the left must be less than the value on the right |
| <= | Comparison | Value on the left must be less than or equal to the value on the right |
| && | Logical | AND – the conditional statement on the left and the right must be true |
| | | | Logical | OR – the conditional statement on the left or the right must be true |
| ! | Logical | NOT – returns the logical opposite of the conditional statement immediately following the operator |

You can mix and match operators to create a Boolean value ‘bool’. Boolean values are either true or false and can be combined with if statements to determine if they should run or be skipped.

## If statements:

This statement says that “if this condition is true than run this block of code” if it isn’t true the program skips this block of code

let temp = 100

if temp >= 100 {

print(“The water is boiling.”)

}

## If-Else Statements:

What if the condition isn’t true? By adding an ‘else’ clause you can specify a block of code to execute if it isn’t true.

} else {

Print(“ the water is not boiling.”)

}

## Boolean values:

You can assign the results of a logical comparison to a Bool constant to check or access the value later. These values can only be true or false.

Let number = 1000

Let isSmallNumber = number < 10

(false)

It is also possible to invert a Bool value using the logical NOT operator:

Var isSnowing = false

If ! isSnowing {

Print(“it’s not snowing.”)

}

## Switch Statement:

A basic switch statement takes a value with multiple options and allows you to run separate code based on each option or ‘case’

Let numberOfWheels = 2

Switch numberOfWheels {

Case 0:

Print(“missing”)

Case 1:

Print(“unicycle”)

Case 2:

Print(“bicycle”)

Etc.

You can also use interval matching to check for inclusion within a range:

Switch distance {

Case 0…9:

Print(“destination near”)

Etc.

## Ternary Conditional Operator:

A common use for an if statement is to set a variable or return a value. If a certain condition is true you want to set a variable to one value and if its false, you want to set a different value.

Many languages include an operator (?:) for writing more concise code.

The ternary operator has three parts:

1. A question with a true or false answer
2. A value if the answer to the question is true
3. A value if the answer to the question is false

Var largest: int

Let a = 15

Let b = 4

Largest = a > b ? a : b

## Control flow quiz:

1. Which of the following logical operators means “equals”?

A: = B: == C: !=D: >=

1. Which of the following logical operators means “NOT”?

A: != B: < C: | | D: !

1. Which of the following logical operators means “greater than”?

A: < B: > C: <= D: >=

1. Which of the following logical operators means “less than or equal to”?

A: < B: > C: <= D: >=

1. Which of the following logical operators means “AND”?

A: ! B: | | C: && D: !=

1. Which of the following logical operators means “OR”?

A: ! B: | | C: && D: !=

1. What will print to the console when executing the following code?

*var time = 6*

*if time < 12 {*

*print(“good morning”)*

*} else if time < 20 {*

*Print(“good afternoon”)*

*}else {*

*Print(“good evening”)*

*}*

A: “good morning” B: “good afternoon” C: “good evening” D: nothing will print

1. What will print to the console when executing the following code?

*var weight = 52*

*if weight <= 50 {*

*print(“have a great flight”)*

*}else{*

*Print(“there is a $25 fee for your luggage”)*

*{*

A: “have a great flight” B: “there is a $25 fee for your luggage” C: nothing

1. What will be the value of grade after executing the following code?

*var grade: Character*

*let score = 78*

*switch score {*

*case 90…100:*

*grade = “a”*

*case 80…89:*

*grade = “b”*

*case: 70…79:*

*grade = “c”*

*default:*

*grade = “d”*

*}*

A: a B: b C: c D: d

1. What is the value of mood after executing the following code?

*let numberOfCookies = 3*

*let mood = number of cookies > 2 ? “☺” : “☹”*

A: numberOfCookies B: 3 C: ☹ D: ☺

# X code

## Locations:

Build and run is in the toolbar area

## Shortcuts:

Command-B: build the project

Command-R: build and run the project

Command-.: stop building or running

Command-/: toggle comments on selected rows of code

Command-[: shift the selected code left

Command-]: shift the selected code right

Command-I: reindent the selected code right

Command-0: reindent the selected code

Option-Command-0: show and hide the inspector area

Why are closures in swift called closures? Closures can capture and store references to any constants and variables from the context in which they're defined. This is known as closing over those constants and variables.

## Xcode quiz:

1. Where is the “build and run” button located in Xcode?

A: debug area B: interface builder C: toolbar D: project navigator

1. If you wanted to add apple pay to your app where would you begin looking?

A: symbol navigator B: capabilities section of the project file C: info.plist D: inspector area

1. What is the keyboard shortcut to move to the top of a file?

A: Command-^ B: Command - < C: Command-B D: Command-U

1. What do you use to switch to running code on a physical device instead of using simulator?

A: Run button B: Scheme menu C: Debug area

1. Which of the following are requirements for running code on an actual device?

A: an apple ID B: registering an email address as a developer account C: adding the developer account to Xcode D: $99 membership

1. After code execution has paused at a breakpoint which button will resume execution?

A: continue B: step over C: step into D: step out

# Unit 2: Introduction to UIKit:

# Strings

String literals are used to set an initial value for a constant or value. If you assign a string to a constant the string can’t be modified. If it’s a value it can be.

You can use “”” to surround a string that is within a paragraph on different lines before and after the string.

To include double quotes in single line string literals you’ll need to sue the backslash ‘\’ as the escape character

You can use the escape character with other letters and symbols to produce specific results:

* Double quote: \”
* Single quote: \’
* Backslash: \\
* Tab: \t
* Newline (go to the next line – like pressing return: \n

## Character:

Individual characters are of the type ‘character’ you will have to specify this:

Let a = “a”

Let b: character = “b”

## Concatenation and interpolation:

You can combine strings using the += operator

Var myString = “hello”

myString = myString + “, world!”

myString += “ Hello!”

you can also insert the raw value of a constant or variable into a string by preceding the name with a backslash and wrapping the name in parenthesis.

let name = “troy”

let age = 30

print(“\(name) is \(age) years old”)

you can also place expressions within the parenthesis.

Print(“this equals \(a+b)”)

## String equality and comparison:

You can compare strings to see if they’re equal by using the == operator, this checks for identical characters in the same order, uppercase characters aren’t equal to lowercase which is known as case sensitivity. If you want to ignore the capitalisation of a string when checking for string equality you can use the ‘lowercased()’ method to normalise the two. (or ‘uppercased()’)

If you want to check if a string is somewhere within another string, you can use the .contains() method to check if the substring is found, such as when you include your name in a password, and it says you can’t contain your name.

You can use switch statements to perform specific blocks of code based on a particular case, or to pattern match multiple values of strings or characters.

## More advanced string topics:

* startIndex
* endIndex
* index(before: )
* index(after: )
* index(\_:offsetBy: )
* insert(\_:at:)
* insert(\_offsetBy:)
* instert(contentsOf:at:)
* remove(at:)
* removeSubrange(\_:)
* replaceSubrange(\_:width:)

## Strings quiz:

1. when you declare a string how do you set the initial value?

A: with Unicode B: with a string literal C: with the operator += D: with an index

1. to compare strings how would you check for equlity?

A: with the operator == B: with a string literal C: with an index D: with the lowercased() method

1. What is the keyboard shortcut to move to the top of a file?

A: Command-^ B: Command - < C: Command-B D: Command-U

1. Why does the following expression return false?

*“Jonathan” == “jonathan”*

A: it doesn’t B: an uppercase and lowercase character aren’t the same thing C: because == is the wrong operator for string equality

# Functions

Argument labels are important to label your functions properly. One example is the label of ‘to’ to add context to your code.

You will sometimes need to define the parameter values, such as writing ‘return’ to specify when a function will return

## Defining a function:

func *functionName* (*parameters*) *-> ReturnType* {

}

## Parameters:

To specify a function with a parameter, insert a name for the value, a colon, and the values type, all inside the brackets. E.g.

Func triple(value: int) {

Let result = value \* 3

## Functions quiz:

1. Which of the following statements are true about the function below?

*func greet(name: String) {*

*print(“hello, \(name)”)*

A: it can be called using greet(“Jason”) B: the first parameter requires an argument label C: the function returns a string D: the function has no return value

1. Why doesn’t this compile?

*func increment(\_value: int by amount: int)-> {*

*value + amount*

*}*

A: no return statement B: needs an argument label C: Increment already a function defined in swift D: function parameters aren’t separated by commas

1. How was the function most likely declared?

*func multiply…...*

*let value = multiply(5, by: 4)*

A: (\_first: int, by: Int) B: (first: int, by: Int) -> Int C: (\_first: int, by: int) -> Int D: (first: int, by: double) -> Int

# Structures

A structure is used to0 define a new type ‘struct’ to use that type you must create an instance of it.

## Instances:

After creating an instance each instance inherits all the properties and features of the structure, e.g.

struct Shirt {

var size: size

var color: color

}

let myShirt = Shirt(size: .xl, color: .blue)

let yourShirt = Shirt(size: .m, color: .red)

this creates 2 separate instances of an individual shirt

## Initializers:

All structures come with at least one initializer. This is like a function that returns a new instance of the type. ‘init()

var string = String.init()

when you define a new type, you need to consider how you’ll create new instances. Another way is just creating default values for your instance properties.

struct odometer {

var count: Int = 0

}

let odometer = Odometer()

print(odometer.count)

= 0

Initializers are the most common way to create new instances of your custom structures

## Instance methods:

These are functions that can be called on specific instances of a type. They provide ways to access and modify properties of the structure and adds functionality that relates to its purpose.

struct Size {

var width: Double

var height: Double

func area() -> Double {

width\*height

}

}

let someSize = Size(width: 10.0, height: 5.5)

let area = someSize. area()

## Mutating methods:

Sometimes you’ll want to update the property values of a structure within an instance method, so you’ll need to add the mutating keyword before the function.

struct Odometer {

var count: Int = 0

mutating func increment() {

count += amount

}

mutating func increment(by amount: Int) {

count += amount

}

mutating func reset() {

count = 0

}

}

var odometer = Odometer() //defaults to 0

odometer.increment() //goes to 1

odometer.increment(by: 15) //goes to 16

odometer.reset() //reset to 0

## Computed properties

There is a feature that allows a property to perform logic that returns a calculated value.

With computed properties you can create properties that can compute their value based on other instance properties or logic.

struct Temperature {

var Celsius: Double

var Fahrenheit: Double {

Celsius \* 1.8 + 32

}

var kelvin: Double {

Celsius + 273.15

}

}

To add a computed property, you must first declare the property as a variable and must declare the type, then you can declare the logic it will return.

## Property observers:

You can observe any property and respond to the changes in the property’s value. These property observers are called every time a property’s value is set. There are two codes. ‘willSet’ and ‘didSet’. Whenever willSet is called first you’ll have access to the new value that will be set to the property value in a constant called newValue After the property’s value has been updated you can access the previous property value using ‘oldValue’

struct StepCounter {

var totalSteps: Int = 0 {

willSet {

print(“\(newValue)”)

}

didSet {

if totalSteps > oldValue {

print(“\(totalSteps – oldValue)”)

}

}

}

}

## Type properties and methods:

Use the ‘static’ keyword to add a property or method to a type. This is useful when a property is related to the type but not a characteristic of it

struct Temp {

static var boilingPoint = 100

}

Type methods are similar to properties, but they are used when the action is related to the type but not something that a specific instance of the type should perform. The Double structure contains a static method named minimum that returns the lesser value.

let smallerNumber = Double.minimum(100.0, - 1000.0)

## Copying:

If you assign a structure to a variable or pass an instance as a parameter into a function to values are copied. Separate variables are therefore separate instances of the value so changing one value won’t change the other.

## Self:

‘self’ refers to the current instance of the type. It can be used within an instance method or computed property to refer to its own instance. It is optional usually as swift is smart.

## Variable properties:

These provide a convenient way to create new data from old data by using one variable as a base to create another variable.

## Structures quiz:

1. When is it useful to define a custom structure

A: when you want to represent simple data already handled by a type B: when you want to represent a new type of data with a collection of properties and functions

1. Which of the following is not a key component of a structure?

A: name B: properties C: functions D: enumerations

1. What do you call a function that’s added to a structure?

A: function B: instance function C: instance method D: type method

# Classes and inheritance

## Class:

They are like structures. Both can define properties to store values and define methods to provide functionality and define initialisers to set up the initial state. You can define class using the ‘class’ keyword.

class Person {

let name: String

init(name: String) {

self.name = name

}

}

## Inheritance:

The biggest difference between structures and classes is that classes can have hierarchal relationships. They can have parent and child classes. A parent is a superclass, and a child is a subclass. Subclasses inherit properties and methods from super classes, but they can also augment or replace the implementation of the methods and properties. A class on its own is known as a base class.

class Bicycle: Vehicle {

} \\ this defines a subclass called a bicycle with a superclass of vehicle

To override a function that would otherwise be inherited you prefix your new definition with the ‘override’ keyword, e.g. ‘override func makeNoise()’

## References:

Classes can reference values assigned to a constant or variable. When you create an instance of a class swift picks out a region in the devices memory to store that instance.

## Memberwise initialisers:

Unlike structure there is not a member wise initializer created for classes they define.

## Class or structure?

All new types should be set up as structures until you need a feature that a class provides

## Classes and inheritance Quiz:

1. Which of the following statements are true about these clauses?

*class Scientist {}*

*class Geologist: Scientist {}*

*class Physicist: Scientist {}*

A: Scientist is a base class B: there is only one subclass C: Geologist and physicist are both descendants of Scientist D: physicist doesn’t inherit from scientist

1. Why doesn’t this compile?

*class Scientist {*

*func performCalculation() {}*

*}*

*class Physicist: Scientist {*

*func performCalculation() {}*

*}*

A: Scientist doesn’t have the initializer needed B: the second function doesn’t call the parent using super C: the second function is missing the override keyword

# Collections

## Arrays:

This stores an ordered list of same-typed values. When you declare an array, you can specify what types of values will be held in the collection

var names: [string] = [“Anne”, “Gary” “Keith”]

## Array types:

An array is like a basket. It can start out empty and can be filled with values later. But if an array literal doesn’t contain any values how can its type be inferred? You can declare the type using type annotation, collection type annotation or an array initializer

Var myArray: [Int] = [ ]

## Working with arrays:

Instead of counting out 100 zeros you can create an array of count 100 with repeating default values:

Var myArray = [int](repeating: 0, count: 100)

Once you’ve defined an array for you can use various methods and properties to access or modify it

Let firstName = names[0]

Names[1] = “paul”

You can use append to add new values

Var names = [“amy”]

Names.append(“joe”)

Names +=[“keith”, “jane”]

Print(names)

You can also get a name to come in at a specific point

Names.insert(“bob”, at: 0)

## Dictionaries:

The second type of collection in swift is a dictionary. This contains a list of keys and their associated value. Each key must be unique. You can set up a dictionary using a dictionary literal:

[Key1: value1, key2: value2, key3: value3]

You can store lists with the name as a key and the ‘score’ as the corresponding value. A dictionary has a count property to determine the number of key-value pairs and an isEmpty p0roperty to determine whether the dictionary has no key value pairs.

Var myDictionary = [String: Int]()

Var myDictionary = Dictionary<String, : Int>()

Var myDictionary = [String: Int] = [:]

## Add/remove/modify a dictionary:

Since the order in a dictionary doesn’t matter there’s no index and no risk of subscripting errors associated with indices. If you want to know if there’s an old value before replacing it use the ‘updateValue(\_:, forKey:) if there was no value, oldValue will be nil.

Let oldValue = scores.updateValue(100, forKey: “Richard”)

## Accessing a dictionary:

The dictionary provides two properties not included in other collection types, you can use keys to return a list of all the keys within a dictionary and values to return a list of all the values.

Var scores = [“Richard”: 500, “Luke”: 400, “Cheryl”: 800]

Let players = Array(scores.keys)

Let points = Array(scores.values)

## Collections quiz:

1. What is the result of executing numberOfLegs [“snake”] = 0?

*var numberOfLegs = [“spider”: 8, “human”: 2, “dog”: 4, “cat”: 4]*

A: nothing. It cant be changed B: snake is added as a key with a value of 0 C: snake is updated to value 0 instead of 4 D: error because two keys have the same value

1. How would you insert “56123” before “29371”?

*var bankAccounts = [18203, 29371, 47290]*

A: bankAccounts.insert(56123, at: 1) B: bankAccounts.append(56123) C: bankAccounts.insert(56123, at: 2) D: bankAccounts[1] = 56123

# Loops

## For loops:

The for-in loop is useful for repeating something a set number of times or for performing work across a collection of values. It executes a set number of statements for each item within a range, sequence or collection.

For index in 1…5 {

Print(“this is number \(index)”)

}

If your result doesn’t need to use the values in the range, you can skip assigning a value to a constant and replace its name with a \_: you can also use the same syntax to iterate over each item in an array

For \_ in 1…3 {

Print(“hello!”)

}

Let names = [“joe”, “Cathy”, “Winston”]

For name in names {

Print(“hello \(name)”)

}

You can also use it to iterate over each character in a string, and use enumerated() to return an index

For (index, letter) in “abcd”.enumerated() {

Print(“\(index): \(letter)”)

}

## While loops:

A while loop will continue to loop until its condition has been met and is no longer true.

Swift checks the condition before each loop is executed which means its possible to skip the loop entirely if the condition is never satisfied.

Var numberOfLives = 3

While numberOfLives > 0 {

playMove()

updateLivesCount()

}

## Repeat-while loops:

This is like a while loop, but this syntax executes the block once before checking the condition

Var steps = 0

Let wall = 2

Repeat {

Print(“step”)

Steps += 1

If steps == wall {

Print(“you’ve hit a wall!”)

Break

}

} while steps < 10

## Control transfer statements:

You may have situations when you want to stop execute of a loop from within the loops body the swift keyword with break the code execution within the loop and start executing any code defined after the loop

There may also be situations in which you want to skip to the next iteration in a loop. While the break keyword will ned the loop entirely, continue will move onto the next iteration.

## Loops quiz:

1. Which of the following are used to loop over a section of code?

A: for, if, while B: while, range, if C: for, while, do D: for, while

1. Which of the following loops through the range 0-9 and ignores the range value?

A: for value in 0…10 B: for value in 0..<10 C: for \_ in 0…10 D: for \_ in 0..<10

1. What is the main distinction between a view and a control?

A: views respond to user interaction and controls display information B: views display in formation and controls respond to user interaction

# Unit 3: navigation and workflow:

# Optionals

## Nil:

Optionals are useful in situations when a value may or may not be present. An optional represents two possibilities, either there is a value, and you can use it or there’s no value at all. Optionals can solve this problem by providing a wrapper around a value that may exist or may not exist yet. You can think of an optional as a box that when opened will either contain. An instance of the expected type or nothing at all (nil)

Struct book {

Var name: string

Var publicationYear: Int?

}

Let FirstDickens = Book(name: “A Christmas Carol”, publicationYear: 1843)

Let secondDickens = Book (name: “David Copperfield”, publicationYear: 1849)

Let thirdDickens = Book(name: “A Tale of Two Cities”, publicationYear: 1859)

Let books = [firstDickens, secondDickens, thirdDickens]

Let UnannouncedBook = Book(name: “Rebels and Lions”, publicationYear: nil)

## Specialising the type of optional:

You can’t create an optional without specifying the type. In most cations you’ll need to use type annotation to specify the type when creating an optional variable or constant.

Var serverResponseCode: Int? = nil

## Working with optional values:

How do you determine whether an optional contains a value? How do you access the value? You could begin by comparing the optional to nil using an if statement if the value is not nil, you can unwrap the value using the force-unwrap operator, ‘!’

If publicationYear !=nil {

Let actualYear = publicationYear!

Print(actualYear)

}

Optional binding unwraps the optional and, if it contains a value, assigns the value to a constant as a non-optional type, making it safe to work with. This approach eliminates the need to continue working with the ambiguity of whether you’re working with a value or with nil.

## Functions and Optionals:

Swift comes with functions that return optional values. If you want to write a function that accepts an optional as an argument, simply update the type in the parameter list. The same is true for a function that returns an optional. Just update the return type.

Func textFromURL (url: URL) -> String?

## Failable initializers:

Any initialiser that might return nil is called a failable initialiser. For greater control and safety, you may want to create your own failable initialisers and define the logic for returning an instance. You can also use optional binding to safely unwrap the value before proceeding to use it.

Struct Toddler {

Var name: string

Var monthsOld: int

Init?(name: string, monthsOld: int) {

If monthsOld < 12 | | monthsOld > 36 {

Return nil

} else {

Self.name = name

Self.monthsOld = monthsOld

}

}

}

Let toddler = Toddler(name: “Joanna”, monthsOld: 14)

If let myToddler = toddler {

Print(“\(myToddler.name) is \(mytoddler.monthsOld) months old”)

} else {

Print(“the age you specified for the toddler is not between 1 and 3 years of age”)

}

## Optional chaining:

Its also possible for an optional value to have optional properties, these are called nested Optionals. Unwrapping nested Optionals can require lots of code. Rather than assign a name to every optional you can conditionally unwrap each property using a construct known as optional chaining. When chaining together Optionals a ? appears before each optional in the chain. If a nil value breaks the chain at any point the if let statement fails. As a result, no value is assigned to the constant and the code inside of the braces never executes. If none of the values are nil, the code inside of the braces executes and the constant has a value.

If let theApartmentNumber =

person.residence? .address? .apartmentNumber {

Print(“he/she lives in apartment number \(theApartmentNumber).”)

}

## Implicitly unwrapped Optionals:

An object can’t be initialized until all its non-optional properties are given a value. In some cases, some properties are nil for only a moment until the value can be specified after initialization.

Class viewController: UIViewCopntroller {

@IBOutlet var label: UILabel!

}

Swift uses the optional ! to unwrap an implicitly unwrapped optional automatically. This allows you to use ‘label’ as if it weren’t an Optional, while allowing the view controller to be in initialized without it. These Optionals should be used in one special case. When you need to initialise an object without supplying the value, but you know you’ll be giving the object a value before any other code tries to access it. It might seem convenient to overuse these to save yourself from using the ‘if let’ syntax but by doing so you’d remove an important safety feature from the language, if you try to access the value of this specific optional and the value is nil, the program will crash.

## Optionals quiz:

1. Which of the following declares a double with a value of 4.2 that can be set to nil later?

A: let height: Double? = 4.2 B: var height: Double = 4.2 C: var height: Double? = 4.2

D: var height: Double? = nil

1. Which of the following code snippets uses valid optional binding syntax?

A: if let dogName = owner.dogs.first {} B: if dogName = owner.dogs.first {} C: if let dogName = owner.dogs.first! {} D: if let dogName == owner.dogs.first {}

1. Is the following initializer failable?

*struct reportCard {*

*var student: String*

*var averageGPA: Double*

*init?(studentName: String, GPA: Double) {*

*self.student = studentName*

*self.averageGPA = gpa*

A: yes the syntax is correct B: yes but shouldn’t be marked as failable as it never returns nil C: no it never returns nil

# Type casting and inspection

## Type casting:

A functions declaration determines the type of data to be returned. if the function can’t return the type, it will return the parent type instead. You can use the as? Operator to try and downcast the value to a more specific type and store it in a new constant. This operation is known as a conditional cast because it casts the instance to the specified type if it’s possible to do so. Use the if-let syntax to check the conditions before converting the type. Theres also a forced form of the type cast operator, ‘as !” this version will force the downcast to the specified type, but if you specify an incorrect type, it will crash the program. When you know that the returned object will be a more specific type you can use the ‘as !’ operator to cast the value.

Let alansDog = fetchPet(for”Alan”) as ! Dog

## Any:

If you want to work with non-specific types, you can use ‘any’ and ‘anyObject’. Any can represent an instance of any type: strings, doubles, functions, etc. anyObject can represent an instance of any class but not a structure.

## Type casting quiz:

1. If a variable can be set to any given structure what is the variables type?

A: is B: as? C: any D: AnyObject

1. When you conditionally downcast from one type to another and store the value in a constant, which combination of keywords is used?

A: as? B: as! C: is D: if let

1. When is it appropriate to use the as! operator?

A: when you need to unwrap an optional B: when you need to convert to an Any type C: when you need to downcast from one type to another on the condition that the type is valid D: when you need to downcast from one type to another and you can guarantee the type is valid

# Guard

If you have multiple if statements nested within one another, you could change this by using the guard statement to clearly communicate to the person reading this code that specific conditions must be met before proceeding instead of creating an if pyramid.

A guard’s else block is executed only of the expression evaluates to false, the opposite to the if statement. With this you can write a function that returns early if it can’t complete its task, by eliminated all the unwanted conditions you can move conditional checks to the top of the function and put the code you expect to run at the bottom.

Func singHappybirthday() {

guard birthdayIsToday else {

Print(“no one’s birthday is today”)

return

}

guard !invitedGuests.isEmpty else {

Print(“just a family party”)

return

}

guard cakeCandlesLit else {

Print(“the cakes candles haven’t been lit”)

return

}

Print(“happy birthday”)

}

## Guard with Optionals:

When you unwrap an optional using ‘if-let’ syntax to bind it to a constant the constant is valid within the braces

If let eggs = goose.eggs {

Print(“the goose laid \(eggs.count) eggs.”)

}

Instead, you can use the guard let to bind the value within an optional to a constant that’s accessible outside the braces. Both if let and ‘guard let’ let you unwrap multiple Optionals in one statement, however doing so with guard makes all unwrapped values available throughout the rest of the function, not just only within the control flow braces.

Guard let eggs = goose.eggs else { return }

Print(“the goose laid \(eggs.count) eggs.”)

Using guard statements to move conditional code is a way to improve the readability pf your programs.

## Guard quiz:

1. What is the purpose of the guard statement?

A: to simplify control flow and communicate intent B: to eliminate invalid parameters early on C: to perform work that can’t be done with an if statement D: all of the above

# Constant and variable scope

Each constant and variable lives within some sort of scope, a place where its visible and accessible. There are two different types of scope: global and local. Any variable declared in global is called a global variable, and local is a local variable.

Global scope refers to code that’s available from anywhere in your program. When you begin declaring variables inside a playground you’re declaring them in global scope. After you declare a variable in one line its now available to each line after it.

However, when you add a par of curly brackets, for a class, structure, function, if or for, etc. the area within the braces defines a new local scope. Any constant or variable declared within the braces is defined in that local scope and isn’t accessible in any other.

Var age = 55

Func printMyAge() {

Print(“my age: \(age)”)

}

Print(age)

printMyAge()

Age is a global variable

## Variable shadowing:

The next example defines a variable called points in two different location, within the local scope and within the for loops local scope. This is called variable shadowing, and it might not be obvious what will happen when the code is executed.

Func printComplexScope() {

Let points = 100

Print(points)

For index in 1…3 {

Let points = 200

Print(“loop \(index): \(points+index)”)

}

Print(points)

}

printComplexScope()

output:

100

Loop 1: 201

Loop 2: 202

Loop 3: 203

100

Variable shadowing is useful because you can reuse the variable within the scoop of the braces. You can also use it to simplify naming unwrapped Optionals from a guard statement.

## Shadowing and initializers:

You can use shadowing to create clean, and easy to read initializers.

Struct Person {

Var name: string

Var age: int

}

Let Tim = Person(name: “Tim”, age: 35)

Print(Tim.name)

Print(Tim.age)

Since name and age are the names of parameters within the function scope, they shadow the name and age properties defined within the person scope.

## Constant and variable scope quiz:

1. What’s the result of this block of code?

*let sum = 99*

*func computeSum(scores: [Int]) -> Int {*

*var sum = 0*

*for score in scores {*

*sum += score*

*}*

*return sum*

*}*

A: compiler error; sum cannot be defined twice in the same scope B: 99 C: 109 D: 0

1. What’s the result of this block of code if you add this to the end?

*let sum =computeSum(scores: [70, 30, 9])*

A: compiler error; sum cannot be defined twice in the same scope B: 99 C: 109 D: 0

1. Why is self placed in front of the property names of this initializer?

*self.name = name*

*self.numEmployees = numEmployees*

*self.mostPopularRide = mostPopularRide*

A: it enables the swift compiler to distinguish between the parameters and properties with the same name B: self is always required in this context C: it enables the reader to distinguish between the parameters and properties with the same name D: self isn’t required

# Enumerations

These define a common type for a group of related values. You define a new enumeration using the keyword enum. The enum defines the type and the case options define the available values allowed within the type. Once you’ve defined the enumeration you can start using it like any other type in swift. Just specify the type along with the value.

Enum CompassPoint {

case north

case east

case south

case west

}

## Control flow:

You can use the same control flow logic used in if and switch statements responding to bool values when working with different case of an enumeration.

## Type safety benefits:

Enumerations are important as they allow you to represent information, such as strings or numbers in a type-safe way.

enum Genre {

case animated, action, romance, documentary, biography, thriller

}

Struct movie {

Var name: string

Var releaseYear: int?

Var genre: Genre

}

Let movie = movie(name: “Wolfwalkers”, releaseYear: 2020, genre: .animated)

This code is much less error prone as the compiler enforces safety by requiring you to choose a case from the genre enumeration when you initialise a new movie. You would use enumerations any time you want to add type safety where you might otherwise use strings or numbers.

## Enumerations quiz:

1. Which of the following would be best represented with an enumeration?

A: names of people in a room B: Political parties C: addresses D: compass degrees

1. Which of the following would be best represented with an enumeration?

A: Hair colours B: T-shirt sizes C: favourite numbers D: vehicle speeds

1. Which of the following would be best represented with an enumeration?

A: car manufactures B: WIFI network names C: Basketball teams D: Shoe brands

# Segues and navigation controllers

Most apps have many scenes for displaying different types of information. Each of these scenes is backed by a separate controller, instance or class. You can use interface builder to add segues, or transitions between different scenes to make the transition as smooth as possible for the user. You can also create special relationships between scenes by including them in a navigation controller.

## Segues:

A segue defines a transition from one view controller to another. You define segues in interface builder by connecting the start and end points.

You can unwind segues as well: whereas a segue transitions to another scene, an unwind segue transitions from the current scene to return to a previously displayed scene.

Select ViewController in the project navigator and add the following method just below the viewDidLoad() function:

@IBAction func unwindToRed(unwindSegue: UIStoryboardSegue) {

}

## Navigation controllers:

Modal segues are the preferred method of transitioning from one context to another within your app. Some situations require a segue from one view controller to a related view controller.

Navigation controllers manage the stack of view controllers and provides the animations when navigating between related views, such as the settings app. It manages hierarchical data structures by creating a stack

# Building simple workflows

## Design principles:

There is a list of six characteristics to keep in mind when designing apps:

1. **Aesthetic integrity**

The appearance and behaviour should make sense for its goals and purpose

1. **Consistency**

This incorporates features and behaviours In a way that people will expect. Whenever possible, its best to use system-provided interface elements, well known, icons, standard text styles, and uniform terminology to deliver a familiar experience.

1. **Direct manipulation**

Users can see the immediate and visible results of their actions, which adds engagement and facilitates understanding.

1. **Feedback**

Users shouldn’t wonder if an app responded to their actions. Ensure that it provides perceptible feedback, in the form of alerts, animations, or other confirmations to let the user know what’s going on.

1. **Metaphors**

Users catch on faster when an apps objects and actions are metaphors for familiar experiences as it makes them feel comfortable and at home, e.g., swiping is a metaphor for swiping through a magazine

1. **User control**

An app should not take control. It might suggest a course of action or warn about dangerous consequences, but the user should always get to have the final say and make the decisions, there should always be an option to cancel an operation.

## Human interface guidelines:

This is a resource for planning and designing apps

## Navigation hierarchy:

A developer’s job is to implement navigation in a way that supports the purpose of your app without distracting the user. It should feel natural and familiar and shouldn’t dominate the interface or draw focus away from the content

Hierarchal navigation:

In this style the user makes one choice per screen until reaching a destination. To navigate to another destination, they must retrace their steps or start over from the beginning and make different choice. E.g. settings or mail.

Flat navigation:

Users switch between multiple content categories. For flat navigation you will typically use a tab bar controller to organise information at the app level. E.g. music and app store

Content driven navigation:

In this style the user moves freely through the content, or the content itself may define the navigation. E.g. games, books, and other immersive apps

Guidelines:

1. **Design an information structure that makes it fast and easy to get content**

Organise your information in a way that requires a. minimum number of taps, swipes and screens

1. **Use standard navigation components**

e.g. tab bars, segmented controls, table views, collection views, and split views as they’re familiar

1. **Use a navigation bar to traverse a hierarchy of data**

The title can display the user’s current position in the hierarchy and the back button makes it easy to return

1. **Use a tab bar to present peer categories of content or functionality**

This lets people quickly and easily switch between categories or modes of operation

1. **Use the proper transition style**

This is a right to left push transition used by a show segue within a navigation controller. However, if the user is switching contexts, use a modal transition