Workshop 1: Introduction to iOS Development

Martin Mikusovic

What will you learn

iOS 13

Swift 5

Xcode 11

About the workshop

- Weekly (roughly) 2-hour session - Thursdays at 6:30pm (BH(S) 2.05)
- Bring your own Mac (the beginning can be done on iPad or Linux, too)
- Bit of theory at the beginning with some practice, but a lot of practice and projects shortly
- Please, answer question that I will ask - be interactive!
- Ask questions at any time, about anything (raise your hand first)
- Download Xcode from the App Store!

All the materials

- https://github.com/mmikusovic/KCL-iOS-workshop-2020

A bit about me

- iOS Developer for 5 years
- Almost 3 years of work experience as an iOS Developer (part-time/ internship)
- 16 apps on the App Store
- 2nd year Computer Science student at King’s College London
- WWDC17 Scholarship Winner

Workshop 1 - Part 0: Native App Development vs Cross-platform

Types

- Native:
  - iOS - Xcode, Swift / Objective-C
  - Android - Android Studio, Kotlin / Java
- Cross-platform:
  - React Native
  - Flutter
  - Ionic
  - Xamarin

Why Native?

- Performance
- UI / UX design
- Gesture interaction
- Security / reliability
- Integration of sophisticated features
- Up to date SDKs and tools

Workshop 1 - Part 1: Introduction to Swift

A little history

Language father

Chris Lattner

LANGUAGE BIRTH

I started work on the Swift Programming Language in July of 2010. I implemented much of the basic language structure, with only a few people knowing of its existence. A few other (amazing) people started contributing in earnest late in 2011, and it became a major focus for the Apple Developer Tools group in July 2013.
— Chris Lattner

LANGUAGE INSPIRATION

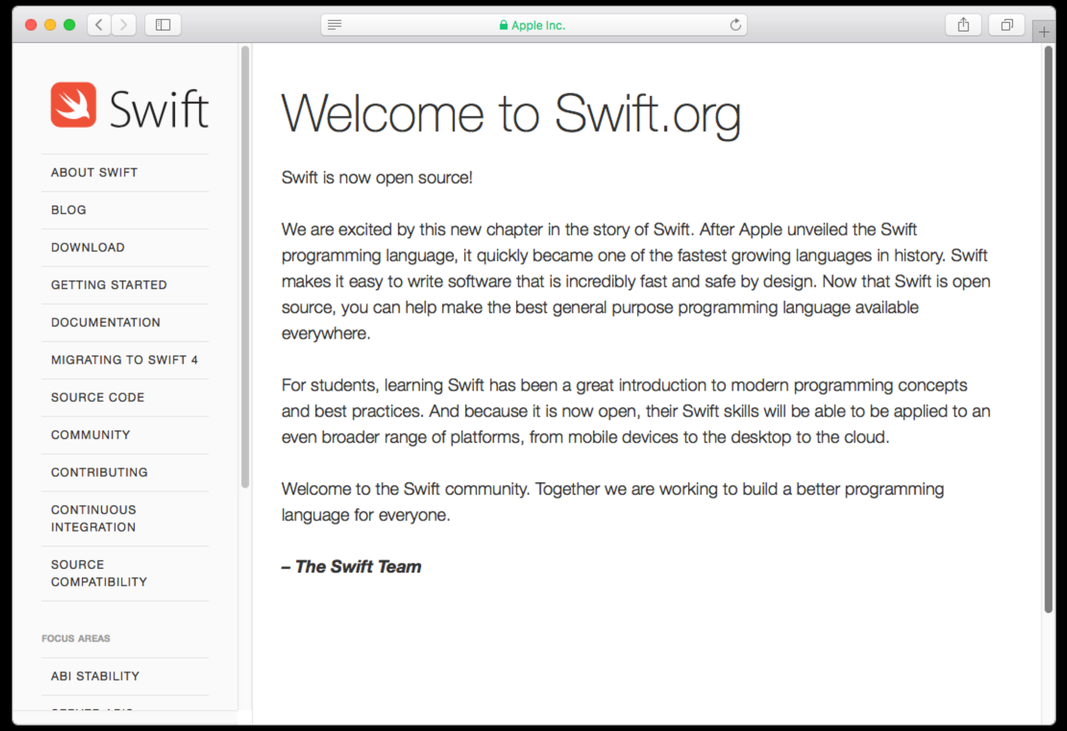
The Swift Programming Language greatly benefited from the experiences hard-won by many other languages in the field, drawing ideas from Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, and far too many others to list.

A modern language

A safe language

- Explicit object "types"
- Type inference
- Optionals
- Error handling

Open Source


Apple Inc.
Swift
ABOUT SWIFT
BLOG
DOWNLOAD
GETTING STARTED
DOCUMENTATION
MIGRATING TO SWIFT 4
SOURCE CODE
COMMUNITY
CONTRIBUTING
CONTINUOUS INTEGRATION
SOURCE COMPATIBILITY
FOCUS AREAS
ABI STABILITY
Welcome to Swift.org
Swift is now open source!
We are excited by this new chapter in the story of Swift. After Apple unveiled the Swift programming language, it quickly became one of the fastest growing languages in history. Swift makes it easy to write software that is incredibly fast and safe by design. Now that Swift is open source, you can help make the best general purpose programming language available everywhere.
For students, learning Swift has been a great introduction to modern programming concepts and best practices. And because it is now open, their Swift skills will be able to be applied to an even broader range of platforms, from mobile devices to the desktop to the cloud.
Welcome to the Swift community. Together we are working to build a better programming language for everyone.
– The Swift Team

Hello, world

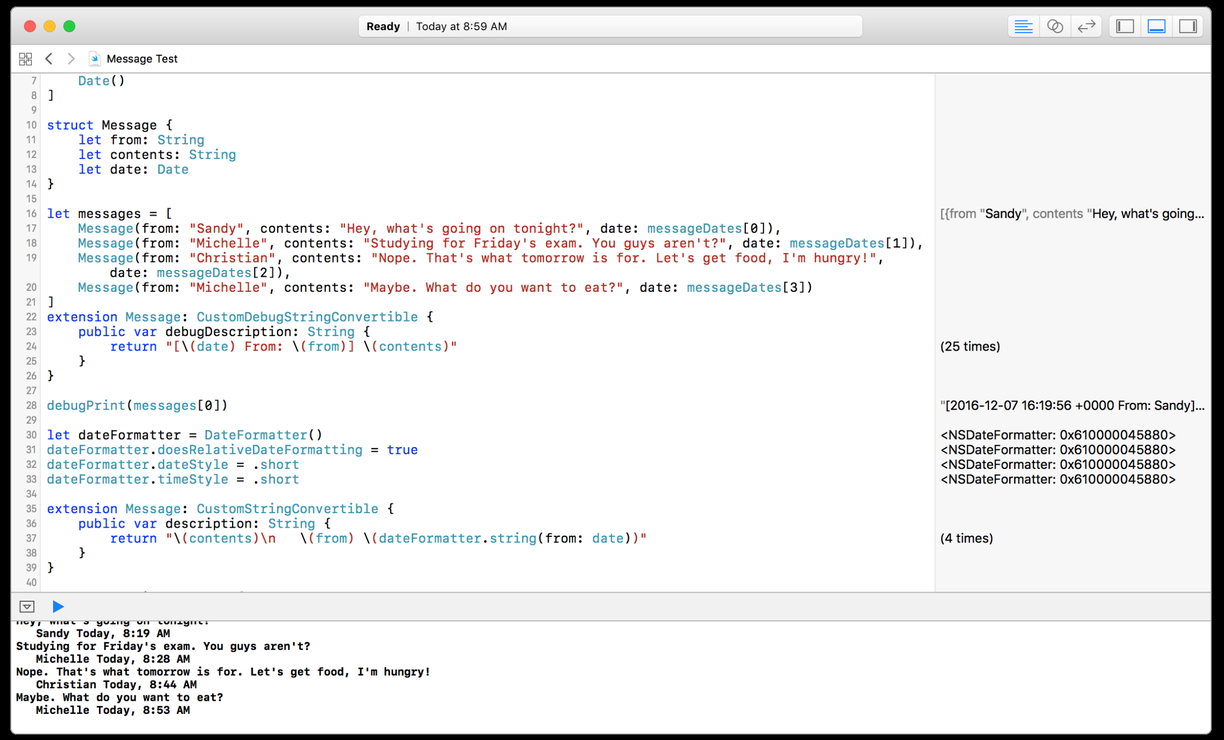
```
print("Hello, world!")
```

main.swift

Hello, world

1. Open Terminal
2. Type `swift` and press Enter
3. Type `print("Hello, world!")` and press Return
4. Type `:quit` and press Return
5. Quit Terminal

Playgrounds



Hello, world

1. Open Xcode
2. Choose File > New > Playground
3. Select iOS, select the Blank template and click Next
4. Name the playground "Hello, world!"
5. Click Create to save the playground
6. Add `print("Hello, world!”)`

Workshop 1 - Part 2:
Constants, Variables, and Data Types

Constants and variables

Associate a name with a value

Defining a constant or variable

- Allocates storage for the value in memory
- Associate the constant name with the assigned value

Constants

Defined using the `let` keyword

```
let name = "John"
```

Defined using the let keyword

```
let pi = 3.14159
```

Can’t assign a constant a new value

```
let name = "John"
name = "James"
```

Variables

Defined using the `var` keyword

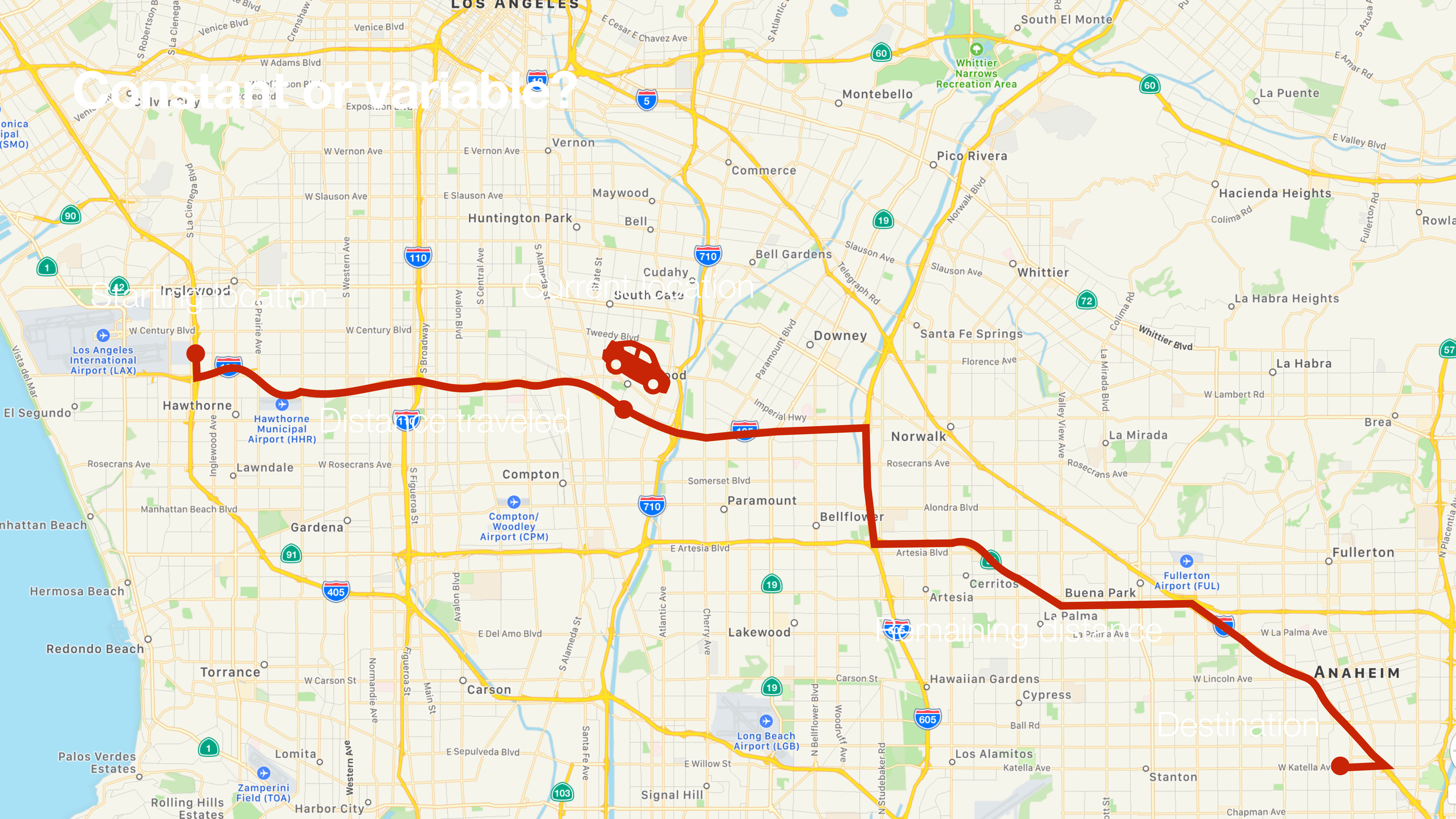
```
var age = 29
```

Can assign a new value to a variable

```
var age = 29

age = 30
```

```
let defaultScore = 100
var playerOneScore = defaultScore
var playerTwoScore = defaultScore

print(playerOneScore)
print(playerTwoScore)

playerOneScore = 200
print(playerOneScore)
```

```
100
100
200
```


Constant or variable?
Starting location
Current location
Distance traveled
Remaining distance
Destination
Los Angeles International Airport (LAX)
Hawthorne Municipal Airport (HHR)
Compton/ Woodley Airport (CPM)
Long Beach Airport (LGB)
Fullerton Airport (FUL)
Zamperini Field (TOA)
Whittier Narrows Recreation Area
LOS ANGELES
Inglewood
Hawthorne
Lawndale
Gardena
Compton
Paramount
Bellflower
Norwalk
Downey
Whittier
Pico Rivera
Montebello
Commerce
Vernon
Maywood
Huntington Park
Bell
Cudahy
South Gate
Bell Gardens
Santa Fe Springs
La Habra
La Habra Heights
La Mirada
Brea
Fullerton
Buena Park
Cerritos
Artesia
Lakewood
Hawaiian Gardens
Cypress
Anaheim
Stanton
Los Alamitos
Carson
Torrance
Redondo Beach
Hermosa Beach
Manhattan Beach
El Segundo
Palos Verdes Estates
Rolling Hills Estates
Harbor City
Lomita
Signal Hill
Hacienda Heights
La Puente
South El Monte
Rowland Heights

Naming constants and variables

Rules

No mathematical symbols (+-*/)

No spaces

Can’t begin with a number

```
let π = 3.14159
let 一百 = 100
let 🎲 = 6
let mañana = "Tomorrow"
let anzahlDerBücher = 15 //numberOfBooks
```

Naming constants and variables

Best practices

1. Be clear and descriptive

   ✗ `n` ✓ `firstName`

2. Use camel case when multiple words in a name

   ✗ `firstname` ✓ `firstName`

Comments

```
// Setting pi to a rough estimate
let π = 3.14
```

```
/* The digits of pi are infinite,
so instead I chose a close approximation.*/
let π = 3.14
```

Types

```
struct Person {
  let firstName: String
  let lastName: String

  func sayHello() {
    print("Hello there! My name is \(firstName) \(lastName).")
  }
}
```

```
struct Person {
  let firstName: String
  let lastName: String

  func sayHello() {
    print("Hello there! My name is \(firstName) \(lastName).")
  }
}

let aPerson = Person(firstName: "Jacob", lastName: "Edwards")
let anotherPerson = Person(firstName: "Candace", lastName: "Salinas")

aPerson.sayHello()
anotherPerson.sayHello()

Hello there! My name is Jacob Edwards.
Hello there! My name is Candace Salinas.
```

Most common types

	Symbol	Purpose	Example
Integer	`Int`	Represents whole numbers	`4`
Double	`Double`	Represents numbers requiring decimal points	`13.45`
Boolean	`Bool`	Represents true or false values	`true`
String	`String`	Represents text	`"Once upon a time…"`

Type safety

```
let playerName = "Julian"
var playerScore = 1000
var gameOver = false
playerScore = playerName
```

Cannot assign value of type ‘String’ to type ‘Int’

```
var wholeNumber = 30
var numberWithDecimals = 17.5
wholeNumber = numberWithDecimals
```

Cannot assign value of type ‘Double’ to type ‘Int’

Type inference

```
let cityName = "San Francisco"
let pi = 3.1415927
```

Type annotation

```
let cityName: String = "San Francisco"
let pi: Double = 3.1415927
```

```
let number: Double = 3
print(number)
```

```
3.0
```

Type annotation

Three common cases

1. When you create a constant or variable before assigning it a value

```
let firstName: String
//...
firstName = "Layne"
```

Type annotation

Three common cases

2. When you create a constant or variable that could be inferred as two or more different types

```
let middleInitial: Character = "J"
var remainingDistance: Float = 30
```

Type annotation

Three common cases

3. When you add properties to a type definition

```
struct Car {
    let make: String
    let model: String
    let year: Int
}
```

Required values

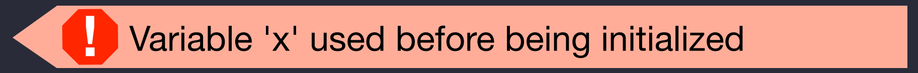
```
var x
```

Required values

```
var x: Int
```

Required values

```
var x: Int
print(x)
```



Required values

```
var x: Int
x = 10
print(x)
```

```
10
```

Numeric literal formatting

```
var largeUglyNumber = 1000000000
var largePrettyNumber = 1_000_000_000
```

Workshop 1 - Part 2
Lab: Variables and Constants

1. Your initial money allocation is 10
2. Your friend's is 20
3. How much money do you have together?
4. Your received 30 (update your allocation accordingly)
5. How much money do you have?

Tuples

LIGHTWEIGHT, TEMPORARY CONTAINERS FOR MULTIPLE VALUES

```
let complex = (1.0, -2.0)     // Compound Type: (Double, Double)
let (real, imag) = complex    // Decompose
let (real, _) = complex       // Underscores ignore value

// Access by index
let real = complex.0
let imag = complex.1

// Name elements
let complex = (real: 1.0, imag: -2.0)
let real = complex.real
```

Optionals

AN OPTIONAL VALUE EITHER CONTAINS A VALUE OR NIL

```
var optionalInt: Int? = 42
optionalInt = nil // to indicate that the value is missing

var optionalDouble: Double? // automatically sets to nil

// Check if nil
optionalDouble == nil
optionalDouble != nil
```

Optionals

FORCE UNWRAP & OPTIONAL BINDING

```swift
let optionalInt: Int? = 42

// Force unwrap
let definitelyInt = optionalInt! // throws runtime error if nil

// Optional binding
if let definitelyInt = optionalInt {
  // runs if optionalInt is not nil and sets definitelyInt: Int
}

guard let definitelyInt = optionalInt else { return }
// runs the rest of the scope if optionalInt is not nil and sets definitelyInt:
Int

// Provide default value
let definitelyInt = optionalInt ?? 0
```

Optionals

IMPLICITLY UNWRAPPED OPTIONALS

```
// Used mainly for class initialization
var assumedInt: Int! // set to nil
assumedInt = 42

var implicitInt: Int = assumedInt // do not need an exclamation mark

assumedInt = nil
implicitInt = assumedInt // ❌ RUNTIME ERROR !!!
```

Enumerations

AN ENUMERATION DEFINES A COMMON TYPE FOR A GROUP OF ITEMS

```
enum CellState {
    case alive
    case dead
    case error
}
let state1 = CellState.alive
let state2 : CellState = .dead // if known type, can drop enumeration name

switch state1 {
case .alive:
    /* ... */
case .dead:
    /* ... */
default:
    /* ... */
}
```

Enumerations

AN ENUMERATION ITEM CAN HAVE AN ASSOCIATED VALUE

```
enum CellState {
    case alive
    case dead
    case error(Int)
}
let errorState = CellState.error(-1)

switch state1 {
case .alive:
    /* ... */
case .dead:
    /* ... */
case .error(let errorCode): // == case let .error(errorCode)):
    /* use errorCode */
}
```

Enumerations

AN ENUMERATION ITEM CAN HAVE A RAW VALUE

```
enum CellState: Int {
    case alive = 1
    case dead = 2
    case error = 3
}

enum CellState: Int { // Specify the Item Raw Value Type
    case alive = 1, dead, error // Int auto-increment
}

let stateValue = CellState.error.rawValue // 3
let aliveState = CellState(rawValue: 1) // alive
```

Classes

COPY REFERENCE ON ASSIGNMENT OR WHEN PASSED INTO FUNCTION

```
// Classes are reference types
class Person {
    var name: String?
    var age: Int = 0
}

// Class reference (not object) are copied when they are passed around
var giuseppe = Person()
let joseph = giuseppe
joseph.name = "Joseph"
giuseppe.name // “Joseph"

// Compare references using === and !==
giuseppe === joseph // true
```

Structures

COPY VALUE ON ASSIGNMENT OR WHEN PASSED INTO FUNCTION

```
// Structures are value types
struct CellPoint {
    var x = 0.0
    var y = 0.0
}

// Structures are always copied when they are passed around
var a = CellPoint(x: 1.0, y: 2.0)
var b = a
b.x = 3.0
a.x // 1.0 – a not changed
```

Structures

STRINGS & CHARACTERS 1/2

```
// String Declaration
var emptyString = "" // == var emptyString = String()
emptyString.isEmpty // true

let constString = "Hello"

// Character Declaration
var character: Character = "p"
for character in "Hello" { /* "H", "e", "l", "l", “o" */ }

// Declare a String as var in order to modify it
var growString = "a"
growString += "b"
growString.append("c")
growString.count // 3
```

Structures

STRINGS & CHARACTERS 2/2

```swift
// String Interpolation
let multiplier = 2
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"

// Print a message in the std output
print(message)

// Comparing Strings
let str1 = "Hello"
let str2 = "Hello"
str1 == str2 // true

str1.hasPrefix("Hel") // true
str2.hasSuffix("llo") // true
```

Structures

Arrays

```
// Array Declaration
var emptyArray = [Int]() // Array<Int>()
var emptyArray : [Int] = []
var array = [25, 20, 16]

// Array Access: subscript out of bounds generate crash
array[1] // 20
array[1000] // ❌ RUNTIME ERROR !!!

array.count // 3
array.isEmpty // false

// Array Iteration
for value in array { /* use value */ }
for (index, value) in array.enumerated() { /* use index and value */ }
```

Structures

VARIABLE ARRAYS

```
var variableArray = [“A"]

variableArray = ["X"] // ["X"]
variableArray[0] = "A" // ["A"]
variableArray.append("B") // ["A", "B"]
variableArray += ["C", "D"] // ["A", "B", "C", “D"]

variableArray.insert("Z", at: 4) // ["A", "B", "C", "D", "Z"]
let a = variableArray.remove(at: 1) // ["A", "C", "D", "Z"]
variableArray[1...2] = ["M"] // ["A", "M", "Z"]
let l = variableArray.removeLast() // ["A", "M"]
```

Structures

CONSTANT ARRAYS

```
let constantArray = [“A"]

constantArray = ["X"] // ❌ ERROR !!!
constantArray[0] = "Y" // ❌ ERROR !!!
constantArray.append("B") // ❌ ERROR !!!
constantArray.remove(at: 0) // ❌ ERROR !!!
```

Structures

DICTIONARIES

```
// Dictionary Declaration
var emptyDictionary = [String: Int]() // Dictionary<String, Int>()
var emptyDictionary: [String: Int] = [:] // key: value
var dictionary = ["First": 25, "Second": 20, "Third": 16]

// Dictionary Access: subscript return Optional
let second: Int? = dictionary["Second"] // 20
let last: Int? = dictionary[“Last"] // nil

dictionary.count // 3
dictionary.isEmpty // false

// Dictionary Iteration
for (key, value) in dictionary { /* use key and value */ }
dictionary.keys // [String]
dictionary.values // [Int]
```

Structures

VARIABLE DICTIONARIES

```
var variableDictionary = ["First" : 25]

variableDictionary = ["Second" : 20] // ["Second" : 20]
variableDictionary["Third"] = 16 // ["Second" : 20, "Third" : 16]

let oldKeyValue = variableDictionary.updateValue(18, forKey: "Second")
// oldValue => 20 // ["Second" : 18, "Third" : 16]

// removes key
variableDictionary["Second"] = nil // ["Third" : 16]
let removedValue = variableDictionary.removeValue(forKey: "Third")
// removedValue => 16 // [:]
```

Workshop 1 - Part 2
Lab: Data types

1. Create enum called Directions with values (south, north, west, east)
2. Create a structure MoveStruct with 1 property called direction of type Directions
3. Create a class MoveClass with the same property
4. Does it compile? What you need to change?

5. Define a String “Hello”
6. Define an Array of Character and append to it each character from the String above
7. Define a Dictionary of integer keys and character values
8. Loop through the array and assign to dictionary index of iteration as a key and value from the array as a value

Property types

LAZY, WEAK AND UNOWNED

```
class Cell {
    lazy var spa = Spa() // lazy (only var): a property whose initial value is not calculated until the first time it is used
}

// Use a weak reference whenever it is valid for that reference to become nil at some point during its lifetime
class Car {
    weak var tenant: Person?
}

// Use an unowned reference when you know that the reference will never be nil once it has been set during initialization.
class City {
    unowned let country: Country
}
```

Property types

PROPERTY ACCESS SCOPE

`open` – access from any source file, enables overriding from everywhere

`public` – access the same as `open`, enables overriding just in the module

`internal` – access in the module (default type)

`fileprivate` – access in the source file

`private` – access from the enclosing declaration and its extensions

`final` – can be added to any except `open`, prevents subclasses from overriding

Extensions

AVAILABLE FOR ENUMERATIONS, STRUCTURES & CLASSES

```
// Extensions can make an existing type conform to a protocol
extension SomeType: SomeProtocol {

    var stored = 1 // ❌ ERROR !!! // Extensions CANNOT add store properties !

    var computed: String { /* ... */ } // Extensions can add computed properties

    func method() { /* ... */ } // Extensions can add methods (type and instance)

    convenience init(parameter: Int) { /* ... */ } // Extensions can add initializers

    enum SomeEnum { /* ... */ } // Extensions can add nested types
}
```

Extensions

EXTENSIONS CAN EXTEND ALSO SWIFT LIBRARY TYPES

```
extension Int {
    func times(_ task: () -> ()) {
        for _ in 0..<self {
            task()
        }
    }
}

3.times({ print("Developer! ") })
// Developer! Developer! Developer!
```

Protocols

AVAILABLE FOR ENUMERATIONS, STRUCTURES & CLASSES

```
// Protocols define a blueprint of requirements that suit a functionality
protocol SomeProtocol {
    // Protocols can require instance properties (stored or computed)
    var instanceProperty: Type { get set }
    // Protocols can require type properties (stored or computed)
    static var typeProperty: Type { get set }
    // Protocols can require instance methods
    func someMethod()
    // Protocols can require instance mutanting methods
    mutating func someMutatingMethod()
    // Protocols can require type methods
    static func someTypeMethod()
    // Protocols can require initializers
    init()
}
```

Protocols

CONFORMING TO A PROTOCOL & PROTOCOL USED AS A TYPE

```
// Conforming to a protocol
class SomeClass: SomeProtocol {
    // init requirements have 'required' modifier
    required init() { /* ... */ }
    // Other requirements do not have 'required' modifier
    var someReadWriteProperty: Type
    ...
}

// Protocol used as type
let thing: SomeProtocol = SomeClass()
let things: [SomeProtocol] = [SomeClass(), SomeClass()]
```

Protocols

DELEGATION

```
protocol GameDelegate {
    func didPlay(game: Game)
}

class Game {
    var delegate: GameDelegate? // Optional delegate

    func play() {
        /* play */
        delegate?.didPlay(game: self) // use Optional Chaining
    }
}
```

Protocols

ANY & ANYOBJECT

```
// Any: any instance of any type
let instance: Any = { (x: Int) in x }
let closure: (Int) -> Int = instance as! (Int) -> Int

// AnyObject: any object of a class type
let instance: AnyObject = Car()
let car: Car = instance as! Car

let cars: [AnyObject] = [Car(), Car(), Car()] // see Array
for car in cars as! [Car] { /* use car: Car */ }
```

Workshop 1 - Part 2
Lab: Data types

1. Create a protocol GameDelegate with function called sayHello
2. Create classes Game and Player
3. Create an optional delegate property in Player and a function hello, where delegate.sayHello() is called
4. Make game conform to GameDelegate and add protocol stubs (print “Hello” in function sayHello)
5. Create properties for Game and for Player
6. Assign Player’s delegate to Game
7. Call function hello() on Player
8. Is “Hello” printed?

Workshop 1 - Part 3: Operators

Assign a value

Use the = operator to assign a value

```
var favoritePerson = "Luke"
```

Use the = operator to modify or reassign a value

```
var shoeSize = 8
shoeSize = 9
```

Basic arithmetic

You can use the `+`, `−`, `*`, and `/` operators to perform basic math functions

```
var opponentScore = 3 * 8
var myScore = 100 / 4
```

You can also use the value of other variables

```
var totalScore = opponentScore + myScore
```

Or you can use the current variable you’re updating

```
myScore = myScore + 3
```

Basic arithmetic

Use Double values for decimal point precision

```
let totalDistance = 3.9
var distanceTravelled = 1.2
var remainingDistance = totalDistance - distanceTravelled
print(remainingDistance)
```

```
2.7
```

Basic arithmetic

```
let x = 51
let y = 4
let z = x / y
print(z)
```

```
12
```

Basic arithmetic

Using Double values

```
let x: Double = 51
let y: Double = 4
let z = x / y
print(z)
```

```
12.75
```

Compound assignment

```
var myScore = 10
myScore = myScore + 3
```

```
myScore += 3
myScore -= 5
myScore *= 2
myScore /= 2
```

Order of operations

1. `(` `)`
2. `*` `/`
3. `+` `-`

```
var x = 2
var y = 3
var z = 5
print(x + y * z)
print((x + y) * z)
```

```
17
25
```

Numeric type conversion

```
let x = 3
let y = 0.1415927
let pi = x + y
```

Numeric type conversion

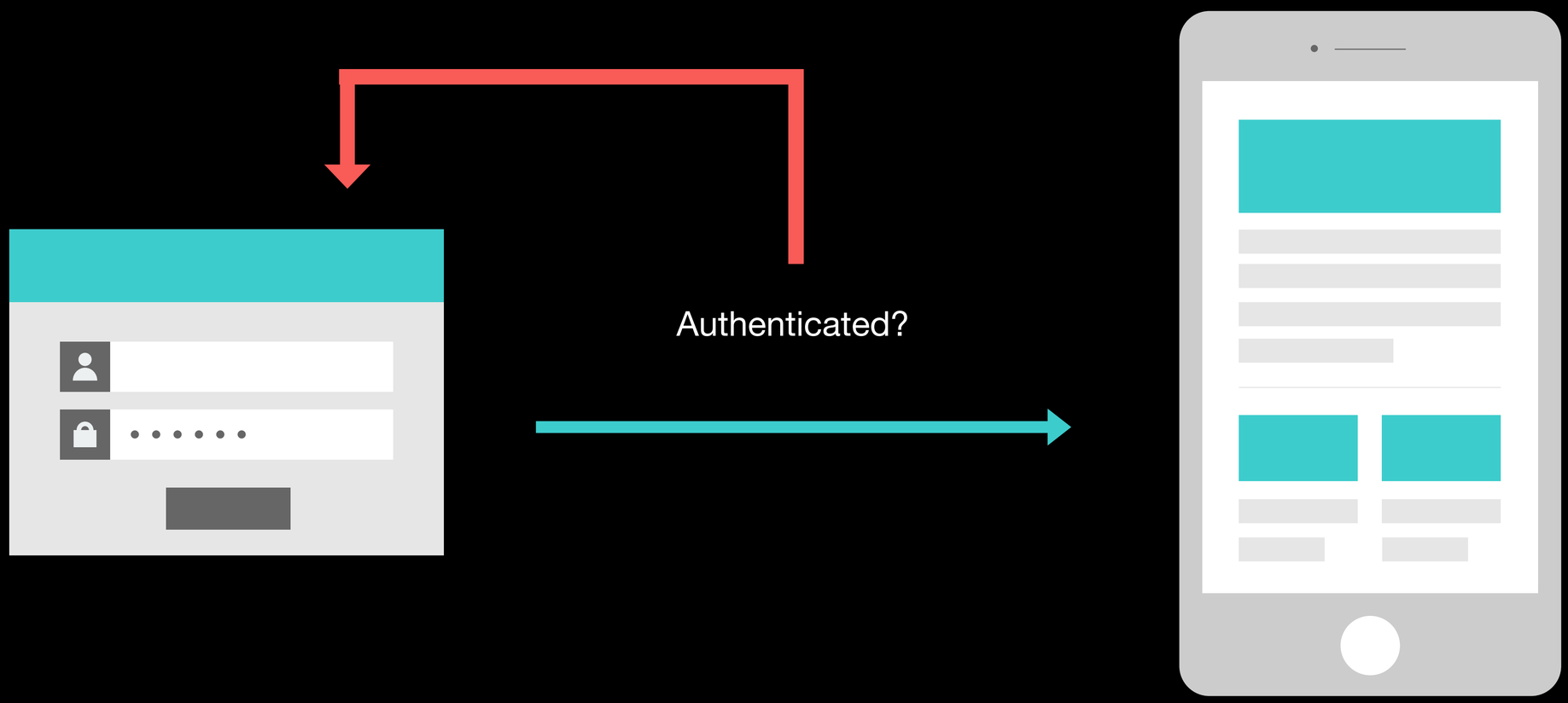
```
let x = 3
let y = 0.1415927
let pi = Double(x) + y
```

Workshop 1 - Part 3
Lab: Operators

1. Create a constant examMultiplier with value 0.85
2. examResult with value 94
3. cwMultiplier with value 0.15
4. cwResult with value 99
5. Print your module final result

Workshop 1 - Part 4:
Control Flow

Conditional flow



Logical operators

Operator	Description
`==`	Two items must be equal
`!=`	The values must not be equal to each other
`>`	Value on the left must be greater than the value on the right
`>=`	Value on the left must be greater than or equal to the value on the right
`<`	Value on the left must be less than the value on the right
`<=`	Value on the left must be less than or equal to the value on the right
`&&`	AND—The conditional statement on the left and right must be true
`||`	OR—The conditional statement on the left or right must be true
`!`	Returns the opposite of the conditional statement immediately following the operator

if statements

```
if condition {
     code
}
```

```
let temperature = 100
if temperature >= 100 {
  print("The water is boiling.")
}
```

```
The water is boiling
```

if-else statements

```
if condition {
    code
} else {
    code
}
```

```
let temperature = 100
if temperature >= 100 {
  print("The water is boiling.")
} else {
  print("The water is not boiling.")
}
```

Boolean values

```
let number = 1000
let isSmallNumber = number < 10
```

```
let speedLimit = 65
let currentSpeed = 72
let isSpeeding = currentSpeed > speedLimit
```

Boolean values

NOT

```
var isSnowing = false
if !isSnowing {
  print("It is not snowing.")
}
```

```
It is not snowing.
```

Boolean values

AND

```
let temperature = 70
if temperature >= 65 && temperature <= 75 {
  print("The temperature is just right.")
} else if temperature < 65 {
  print("It's too cold.")
} else {
  print("It's too hot.")
}
```

```
The temperature is just right.
```

Boolean values

OR

```
var isPluggedIn = false
var hasBatteryPower = true
if isPluggedIn || hasBatteryPower {
  print("You can use your laptop.")
} else {
  print("😱")
}
```

switch statement

```
switch value {
case n:
   code
case n:
   code
case n:
   code
default:
   code
}
```

```
let numberOfWheels = 2
switch numberOfWheels {
case 1:
    print("Unicycle")
case 2:
    print("Bicycle")
case 3:
    print("Tricycle")
case 4:
    print("Quadcycle")
default:
    print("That's a lot of wheels!")
}
```

switch statement

Multiple conditions

```
let character = "z"

switch character {
case "a", "e", "i", "o", "u" :
    print("This character is a vowel.")
default:
    print("This character is not a vowel.")
}
```

switch statement

Ranges

```
switch distance {
case 0...9:
    print("Your destination is close.")
case 10...99:
    print("Your destination is a medium distance from here.")
case 100...999:
    print("Your destination is far from here.")
default:
    print("Are you sure you want to travel this far?")
}
```

switch challenge

Rewrite the following using a switch statement:

```
let temperature = 70
if temperature >= 65 && temperature <= 75 {
  print("The temperature is just right.")
} else if temperature < 65 {
  print("It's too cold.")
} else {
  print("It's too hot.")
}
```

Hint: The smallest possible value for an integer is `Int.min`

switch challenge

Solution

```
let temperature = 76
switch temperature {
case Int.min...64:
    print("It's too cold.")
case 65...75:
    print("The temperature is just right.")
default:
    print("It's too hot.")
}
```

Ternary operator

```
var largest: Int
let a = 15
let b = 4

if a > b {
  largest = a
} else {
  largest = b
}
```

Ternary operator

?:

```
variable = condition ? true_value : false_value
```

```
var largest: Int
let a = 15
let b = 4

largest = a > b ? a : b
```

Workshop 1 - Part 4
Lab: Control Flow

1. Create two constants with some integer values
2. Create a variable largest of type integer
3. Assign to variable largest, the larger from these 2 numbers using if statement and print largest
4. Do the same using ternary operator
5. Which approach do you like more?

Workshop 1 - Part 5:
Loops and functions

Loops

FOR [IN]

```
// old school c-style for loop
for var index = 0; index < 2; index++ { /* use index */ }

// new iterator-style for-in loop
for value in 0..<2 { /* use value */ }
```

Loops

[DO] WHILE

```
// while evaluates its expression at the top of the loop
while x > 2 { /* */ }

// do-while evaluates its expression at the bottom of the loop
do { /* executed at least once */ } while x > 2
```

Functions

DECLARATION & CALL

```
// With parameters and return value
func foo(parameter1: Type1, parameter1: Type2) -> ReturnType {
    /* function body */
}
foo(parameter1: argument1, parameter2: argument2) // call

// Without parameters and return value
func bar() -> Void { /* function body */ }
func baz() -> () { /* function body */ }
func quz() { /* function body */ }
quz() // call
```

Functions

EXTERNAL, LOCAL & DEFAULT PARAMETERS

```
// external and local parameter names
func foo(externalParameterName localParameterName: Type) {
    /* body use localParameterName */
}
foo(externalParameterName: argument) // call must use externalParameterName label

// _ = shorthand for external parameter names
func bar(_ parameterName: Type) { /* body use parameterName */ }
bar(argument) // call has no parameter name

// default parameter values
func baz(parameter1: Type1, parameterWithDefault: Int = 42) { /* ... */ }
baz(parameter1: argument1) // call can omit value for parameterWithDefault
baz(parameter1: argument1, parameterWithDefault: 10) // call can modify value for
parameterWithDefault
```

Functions

VARIADIC PARAMETERS

```
func foo(parameter: Int...) {
    // use parameter as an array with type [Int]
}

foo(parameter: 1, 2, 3, 4, 5) // call with multiple arguments
```

Functions

FUNCTION TYPE

```
// The Type of addOne function is (Int) -> Int
func addOne(n: Int) -> Int { return n + 1 }

// Function as parameter
func foo(functionParameter: (Int) -> Int) { /* */ }
foo(functionParameter: addOne)

// Function as return type
func bar() -> (Int) -> Int {
    func addTwo(n: Int) -> Int { return n + 2 }
    return addTwo
}
```

Closures

MEANING & SYNTAX

Closures are blocks of functionality that can be passed around

```
{ (parameter1: Type1, parameter2: Type2) -> ReturnType in
    / * ... */
}
```

Closures

SORTING WITHOUT CLOSURES

```
let array = [1, 2, 3]
func backwards(i1: Int, i2: Int) -> Bool { return i1 > i2 }

var reversed = array.sorted(by: backwards)
```

Closures

SORTING WITH CLOSURES 1/2

```
// Fully declared closure
reversed = array.sorted { (i1: Int, i2: Int) -> Bool in
    return i1 > i2
}

// Infer closure type
reversed = array.sorted { (i1, i2) in return i1 > i2 }

// Implicit returns
reversed = array.sorted { (i1, i2) in i1 > i2 }
```

Closures

SORTING WITH CLOSURES 2/2

```
// Shorthand argument names
reversed = array.sorted(by: { $0 > $1 } )

// Operator functions
reversed = array.sorted(by: >)

// Trailing closure: outside of () only if it's function’s final
argument
reversed = array.sorted { $0 > $1 }

// Optional parentheses
array.map({ $0 + 1 })
array.map { $0 + 1 } // equivalent
```

Workshop 1 - Part 5
Lab: Loops and functions

1. Define a function loopFor with one integer parameter that will print numbers from 0 until the given number (not including it)
2. Do the same with function loopWhile
3. Call these to functions with the same number
4. Did you get wanted results?

Thank you for your attention