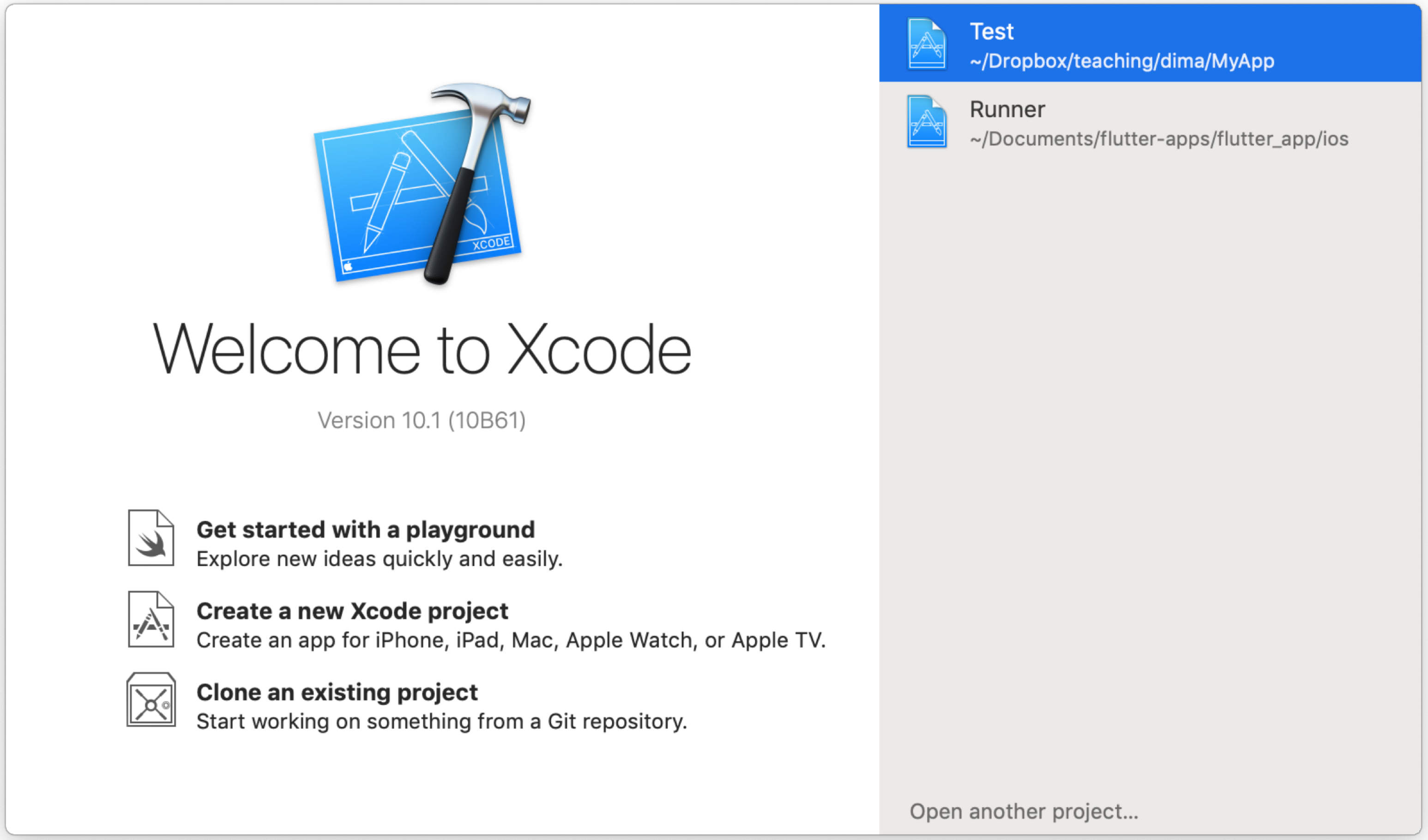
iOS

How to develop apps for iPhones and iPads

Development Environment



Welcome to Xcode

Version 10.1 (10B61)

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.

Clone an existing project
Start working on something from a Git repository.

Test
~/Dropbox/teaching/dima/MyApp

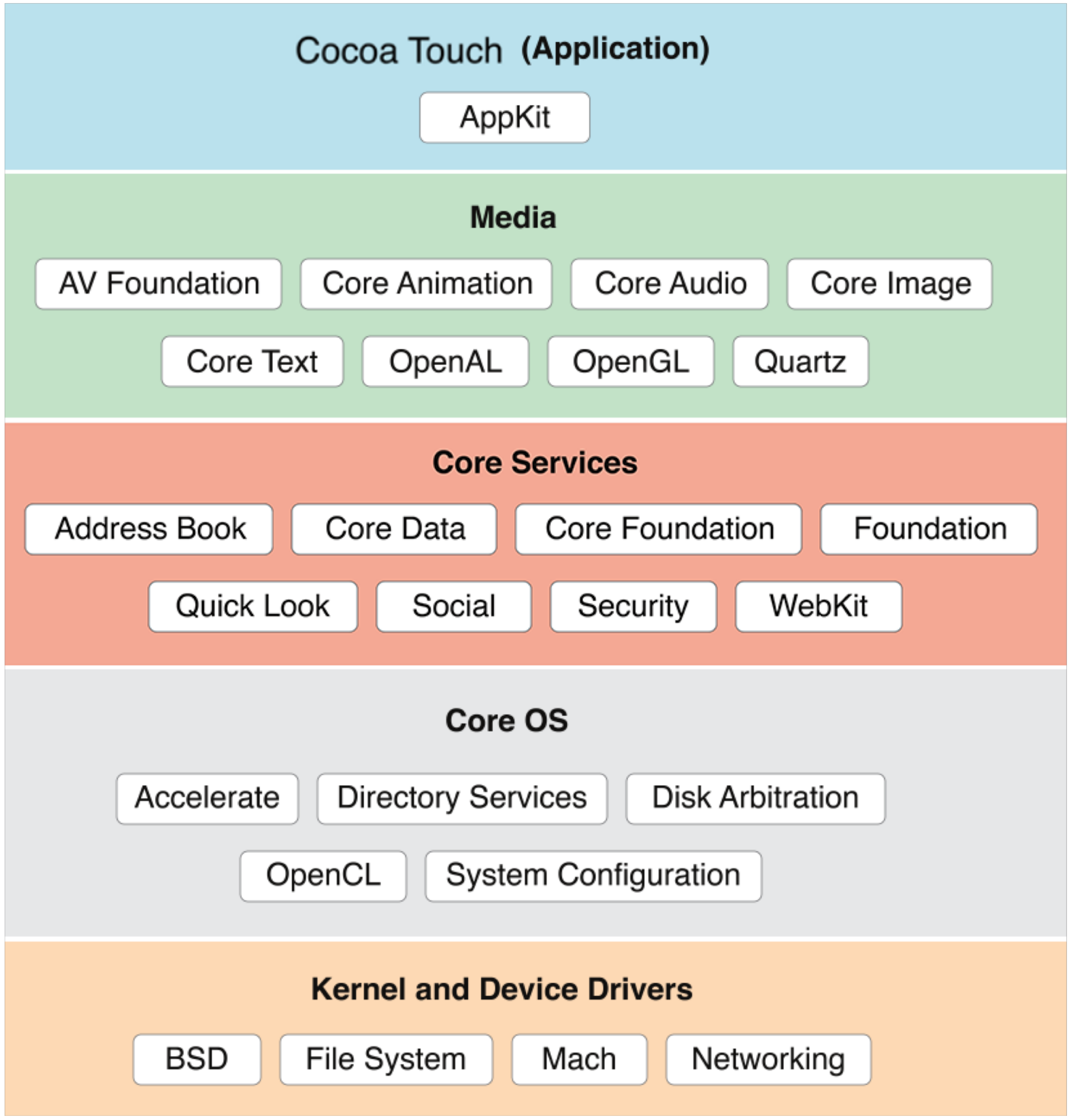
Runner
~/Documents/flutter-apps/flutter_app/ios

Open another project...

Development Environment

- Platforms
  - iOS
  - WatchOS
  - TvOS
- Tools
  - Xcode 10.1 (emulator)
- Language
  - Swift 4
  - Swift Playgrounds

iOS


Cocoa Touch (Application)
AppKit
Media
AV Foundation
Core Animation
Core Audio
Core Image
Core Text
OpenAL
OpenGL
Quartz
Core Services
Address Book
Core Data
Core Foundation
Foundation
Quick Look
Social
Security
WebKit
Core OS
Accelerate
Directory Services
Disk Arbitration
OpenCL
System Configuration
Kernel and Device Drivers
BSD
File System
Mach
Networking

Swift 4

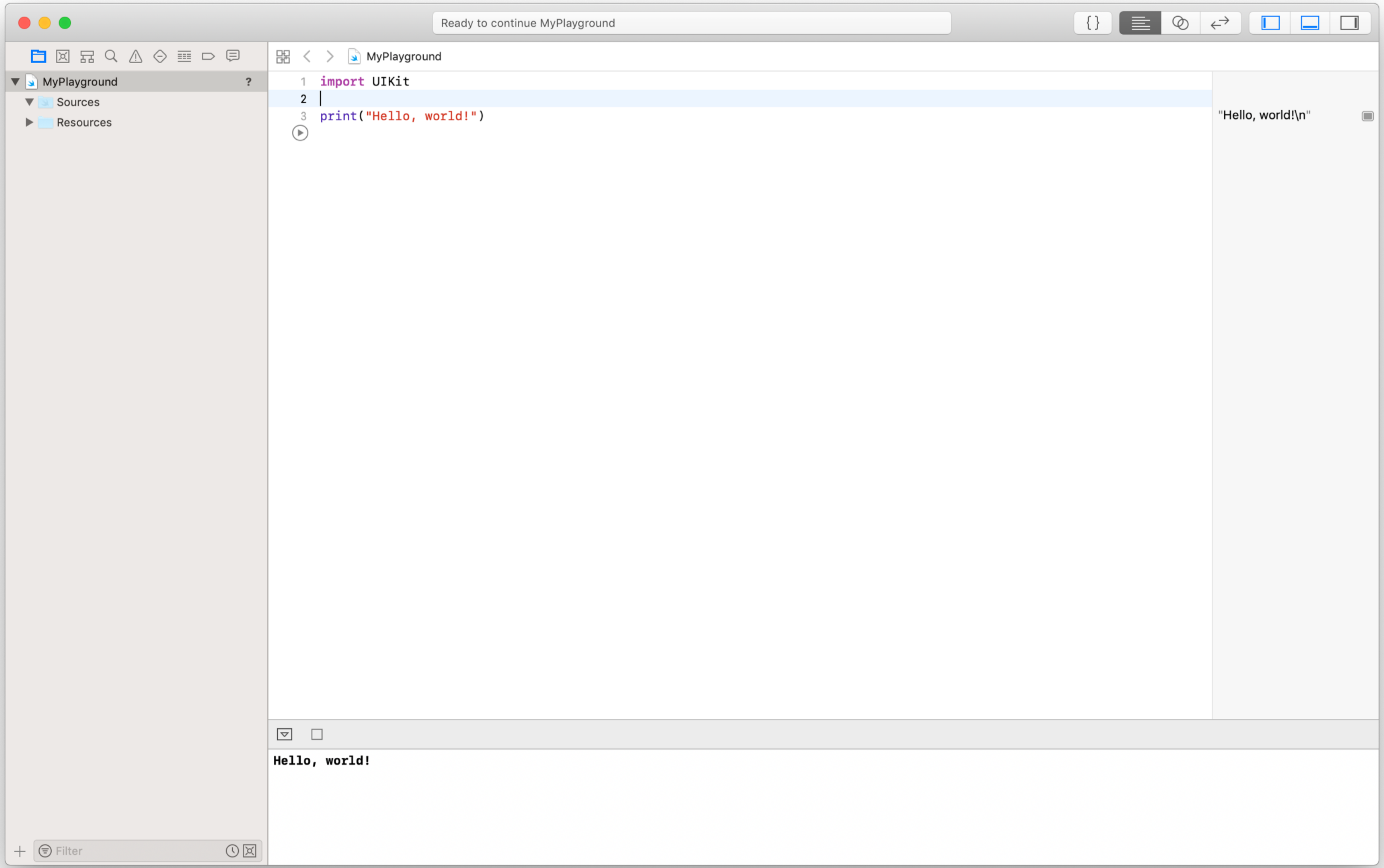
Swift

- Swift features several programming constructs from other languages including ObjectiveC and Smalltalk
- Swift is a multi-paradigm programming environment
- Swift allows type inference e.g. loose typing
- Swift programming statements do not require line ending characters (semicolons)

Our first application

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

Playground



Ready to continue MyPlayground

MyPlayground

Sources

Resources

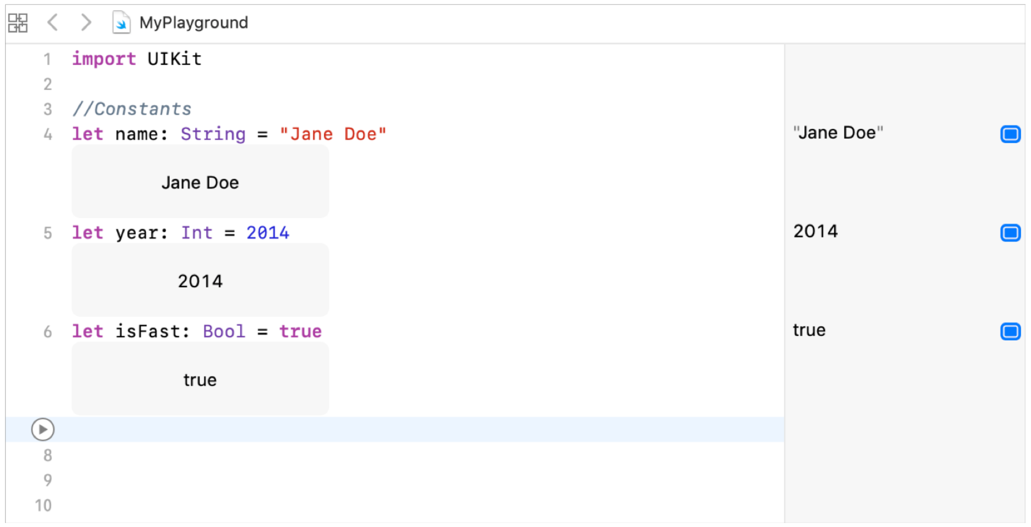
```swift
import UIKit

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

"Hello, world!\n"

```
Hello, world!
```

Variables and constants

```swift
//Variables
var name: String = "Jane Doe"
var year: Int = 2014
var isFast: Bool = true

//Data type detection
var name = "Jane Doe"
var year = 2014
var isFast = true

//Constants
let name: String = "Jane Doe"
let year: Int = 2014
let isFast: Bool = true
```



MyPlayground

```swift
import UIKit

//Constants
let name: String = "Jane Doe"
let year: Int = 2014
let isFast: Bool = true
```

```
"Jane Doe"
Jane Doe
2014
2014
true
true
```

Type Safety and Type Inference

- Swift is type safe
  - Encourages you to be clear about types
  - It performs type checks during compilation
- You don’t have to specify the types explicitly
  - The type is inferred at compilation time depending on the assignments that are made
  - Fewer declarations than with C and Objective-C
  - It’s a good idea to initialize variables when we declare them

Tuples

```
let http404Error = (404, "Not Found")
let (statusCode, statusMessage) = http404Error

print("The status code is \(statusCode)")
print("The status message is \(statusMessage)")

let (justTheStatusCode, _) = http404Error
print("The status code is \(justTheStatusCode)")

print("The status code is \(http404Error.0)")
print("The status message is \(http404Error.1)")

let http200Status = (statusCode: 200, description: "OK")

print("The status code is \(http200Status.statusCode)")
print("The status message is \(http200Status.description)")
```

Optionals

- Use optionals where a value may be absent
  - There is a value and it equals to x or there is no value (nil)

```swift
let posNum = "123"
let convNum: Int? = Int(posNum)

if convNum != nil {
    print("convNum has an integer value of \(convNum!).")
}

// Without ! it would print “convNum has an integer value of Optional(123)”.


let possibleString: String? = "An optional string."
let forcedString: String = possibleString!

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString
```

Operators

- Nil Coalescing Operator
  - (a ?? b) is equivalent to a != nil ? a! : b
  - if the optional a contains a value we take the unwrapped value, if not we take a default b
- Closed Range Operator
  - (a…b) defines a range that goes from a to b (included)
- Half-Open Range Operator
  - (a..<b) defines a range that goes from a to b (excluded)

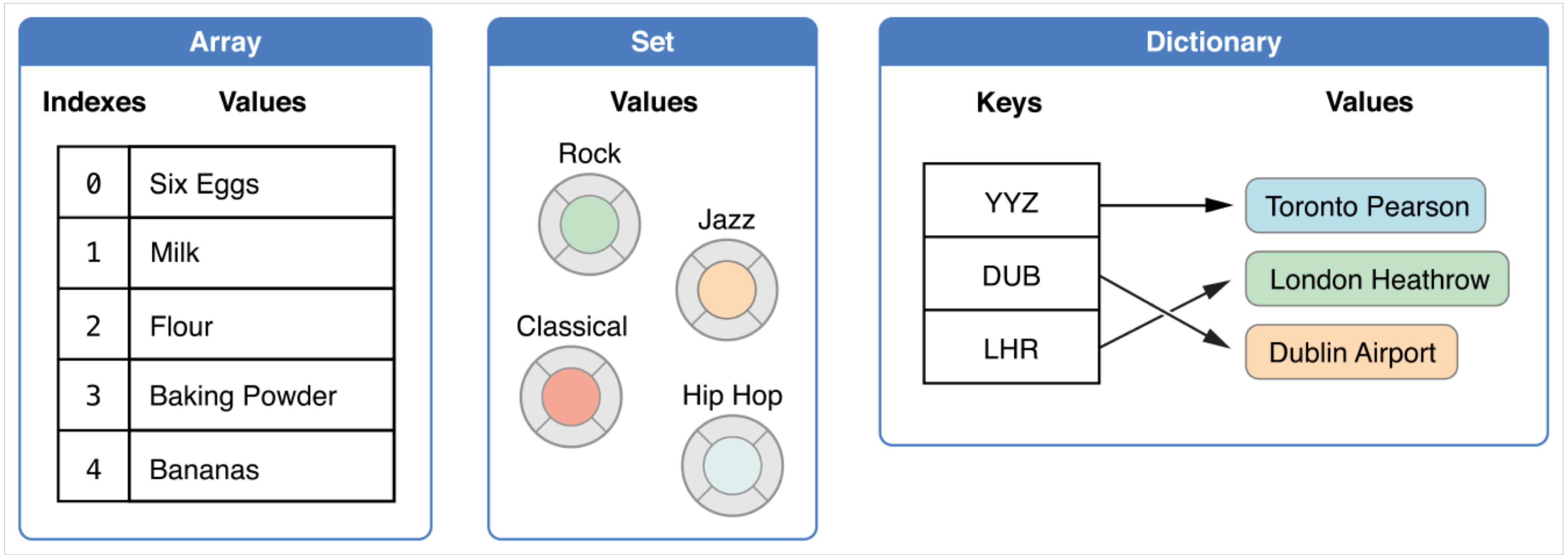
Strings

- Strings are represented by type String

```
for character in "Dog!" {
    print(character)
}
```

  - String mutability depends on let/var
  - Support for String interpolation
- Swift’s type String is a value type
  - A String value is copied when it is passed to a function or method, or when it is assigned to a constant or variable
  - A new copy of the existing String value is created, and the new copy is passed or assigned, not the original version

Collections



Arrays

```swift
var shoppingList: [String] = ["Eggs", "Milk"]
// or var ShoppingList = ["Eggs", "Milk"]

print("The shopping list contains \(shoppingList.count) items.")

if shoppingList.isEmpty {
    print("The shopping list is empty")
} else {
    print("The shopping list is not empty")
}

shoppingList += ["Baking Powder"]
// shoppingList now contains 3 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 6 items

shoppingList.insert("Maple Syrup", at: 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list

shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList now contains 6 items

var firstItem = shoppingList[0]

var someInts = [Int]()
print("someInts is of type [Int] with \(someInts.count) items.")
```

Ways to iterate over arrays

```swift
for item in shoppingList {
    print(item)
}

for (index, value) in shoppingList.enumerated() {
   print("Item \(index + 1): \(value)")
}
```

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

print("I have \(favoriteGenres.count) favorite music genres.")
// Prints "I have 3 favorite music genres."

favoriteGenres.insert("Jazz")
// favoriteGenres now contains 4 items

if let removedGenre = favoriteGenres.remove("Rock") {
    print("\(removedGenre)? I'm over it.")
} else {
    print("I never much cared for that.")
}
// Prints "Rock? I'm over it."

if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
} else {
    print("It's too funky in here.")
}
// Prints "It's too funky in here."

for genre in favoriteGenres {
   print("\(genre)")
}
```

Sets

Set operations

```
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```

Dictionaries

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

print("The airports dictionary contains \(airports.count) items.")

if airports.isEmpty {
    print("The airports dictionary is empty.")
} else {
    print("The airports dictionary is not empty.")
}

airports["LHR"] = "London"
airports["LHR"] = "London Heathrow"
airports["LHR"] = nil //removing an existing value!

if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {
    print("The old value for DUB was \(oldValue).")
}
```

Ways to iterate over a Dictionary

```
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}

for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}

for airportName in airports.values {
    print("Airport name: \(airportName)")
}
```

For-in

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}


let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
print("\(base) to the power of \(power) is \(answer)")

let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
```

While loops

```
while condition {
    statements
}


repeat {
    statements
} while condition
```

Switch

```swift
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
"n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
default:
    print("\(someCharacter) is not a vowel or a consonant")
}
```

. . . with intervals

```swift
let count = 3000000000000
let countedThings = "stars in the Milky Way"
var naturalCount = ""
switch count {
case 0:
    naturalCount = "no"
case 1...3:
    naturalCount = "a few"
case 4...9:
    naturalCount = "several"
case 10...99:
    naturalCount = "tens of"
case 100...999:
    naturalCount = "hundreds of"
case 1000...999999:
    naturalCount = "thousands of"
default:
    naturalCount = "millions and millions of"
}
print("There are \(naturalCount) \(countedThings).")
```

… and with tuples

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("(0, 0) is at the origin")
case (_, 0):
    print("(\(somePoint.0), 0) is on the x-axis")
case (0, _):
    print("(0, \(somePoint.1)) is on the y-axis")
case (-2...2, -2...2):
    print("(\(somePoint.0), \(somePoint.1)) is inside the box")
default:
    print("(\(somePoint.0), \(somePoint.1)) is outside of the box")
}

let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
```

Functions (I)

```swift
func sayHello(personName: String) -> String {
    return "Hello again, " + personName + "!"
}

print(sayHello(personName: "Anna"))

func halfOpenRangeLength(start: Int, end: Int) -> Int {
    return end - start
}

print(halfOpenRangeLength(start: 4, end: 9))

func sayHelloWorld() -> String {
    return "hello, world"
}

print(sayHelloWorld())

func sayGoodbye(personName: String) {
    print("Goodbye, \(personName)!")
}

sayGoodbye(personName: "Anna")
```

Functions (II)

```swift
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
          currentMin = value
        } else if value > currentMax {
          currentMax = value
      }
   }
    return (currentMin, currentMax)
}



let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Prints "min is -6 and max is 109"
```

Functions (III)

```swift
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
          currentMin = value
        } else if value > currentMax {
          currentMax = value
      }
   }
    return (currentMin, currentMax)
}

if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
```

Argument labels

```
func sayHello(person: String, anotherPerson: String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}

print(sayHello(person: "Bill", anotherPerson: "Ted"))

func sayHello(to person: String, and anotherPerson: String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}

print(sayHello(to: "Bill", and: "Ted"))

func sayHello(_ person: String, _ anotherPerson: String) -> String {
    return "Hello \(person) and \(anotherPerson)!"
}

print(sayHello("Bill", "Ted"))
```

Trying to change the value of a function parameter from within the body of that function results in a compile-time error

Default values and variadic parameters

```
func someFunction(parameterWithDefault: Int = 12) {
    // function body goes here
}

someFunction(parameterWithDefault: 6) // parameterWithDefault is 6
someFunction() // parameterWithDefault is 12



func arithmeticMean(numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
       total += number
   }
    return total / Double(numbers.count)
}
print(arithmeticMean(numbers: 1, 2, 3, 4, 5))
// returns 3.0, which is the arithmetic mean of these five numbers
print(arithmeticMean(numbers: 3, 8.25, 18.75))
// returns 10.0, which is the arithmetic mean of these three numbers
```

In-out parameters

- Variable parameters, as described above, can only be changed within the function itself
  - Some limitations

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
```

Function Types (I)

- Every function has a specific function type, made up of the parameter types and the return type of the function

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

var mathFunction: (Int, Int) -> Int = addTwoInts

print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"

mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"

let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// Prints "Result: 8"

func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}

func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}

var currentValue = 3
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function

print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
```

Nested Functions

- All of the functions so far have been examples of global functions, defined at a global scope
- Nested functions are functions inside the bodies of other functions
  - They are hidden from the outside world by default, but can still be called and used by their enclosing function
  - An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope

Example

```swift
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
```

Closures

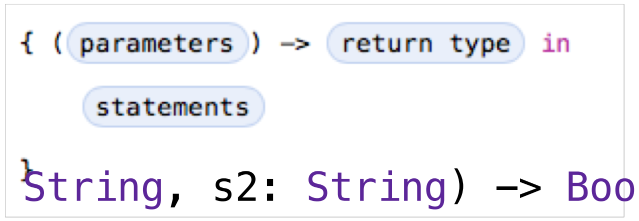
- Closures are self-contained blocks of functionality that can be passed around and used in your code
  - Closures in Swift are similar to blocks in C and to lambdas in other programming languages
- Closures take one of three forms
  - Global functions are closures that have a name and do not capture any values
  - Nested functions are closures that have a name and can capture values from their enclosing function
  - Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context

Closure expressions

- Nested functions are a convenient means of naming and defining self-contained blocks of code as part of a larger function

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

- Closure expression syntax



```
{ (parameters) -> return type in
    statements
}
```

```
var reversedNames = names.sorted(by: {(s1: String, s2: String) -> Bool in
return s1 > s2})
```

Closures

- Inferring Type from Context

```
reversedNames = names.sorted(by: {s1, s2 in return s1 > s2})
```

- Implicit Returns
  - Single-expression closures can implicitly return the result of their single expression

```
reversedNames = names.sorted(by: {s1, s2 in s1 > s2})
```

- Shorthand Argument Names

```
reversedNames = names.sorted(by: {$0 > $1})
```

- Operator Functions
  - Swift's String type defines its string-specific implementation of the greater-than operator

```
reversedNames = names.sorted(by: >)
```

- Trailing Closures

```
reversedNames = names.sorted() {$0 > $1}
reversedNames = names.sorted {$0 > $1}
```

Trailing closures

```swift
func someFunctionThatTakesAClosure(closure: () -> Void) {
    // function body goes here
}

// Here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure(closure: {
    // closure's body goes here
})

// Here's how you call this function with a trailing closure instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

Capturing Values

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
   }
    return incrementer
}

let incrementByTen = makeIncrementer(forIncrement: 10)

incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30

let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()
// returns a value of 7

incrementByTen()
// returns a value of 40
```

Classes and Structures

- Classes and structures can
  - Define properties to store values
  - Define methods to provide functionality
  - Define subscripts to provide access to their values using subscript syntax
  - Define initializers to set up their initial state
  - Be extended to expand their functionality beyond a default implementation
  - Conform to protocols to provide standard functionality of a certain kind
- Classes have additional capabilities that structures do not:
  - Inheritance enables one class to inherit the characteristics of another
  - Type casting enables you to check and interpret the type of a class instance at runtime
  - Deinitializers enable an instance of a class to free up any resources it has assigned
  - Reference counting allows more than one reference to a class instance

Conditions for using structures

- They are used to encapsulate a few relatively simple data values
- Encapsulated values will be copied rather than referenced when you assign or pass around an instance of that structure
- Any properties stored by the structure are themselves value types, which would also be expected to be copied rather than referenced
- A structure does not need to inherit properties or behavior from another existing type

- Many basic data types such as String, Array, and Dictionary are implemented as structures in Swift
  – This means that they are copied when assigned

First example

```swift
struct Resolution {
    var width = 0
    var height = 0
}

class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}

let someResolution = Resolution()
let someVideoMode = VideoMode()
```

- This creates a new instance of the class or structure, with properties initialized to their default values

```swift
print("The width of someResolution is \(someResolution.width)")
print("The width of someVideoMode is \(someVideoMode.resolution.width)")

someVideoMode.resolution.width = 1280
print("The width of someVideoMode is now \(someVideoMode.resolution.width)")
```

Structures

- All structures have an automatically-generated memberwise initializer

```
let vga = Resolution(width: 640, height: 480)
```

- Structures are value types
  - Any structure instance is always copied when it is passed around the code

```
let hd = Resolution(width: 1920, height: 1080)
var cinema = hd //this is a copy: cinema and hd are two different instances

cinema.width = 2048
print("cinema is now \(cinema.width) pixels wide")
// prints "cinema is now 2048 pixels wide"
print("hd is still \(hd.width) pixels wide")
// prints "hd is still 1920 pixels wide"
```

Classes

- Classes are reference types
  - they are not copied, but a reference to the same instance is made

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
tenEighty.frameRate = 25.0

let alsoTenEighty = tenEighty
alsoTenEighty.frameRate = 30.0

print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
// prints "The frameRate property of tenEighty is now 30.0"
```

- Identical to (===), not identical to (!==), and equal (==)

```
if tenEighty === alsoTenEighty {
    print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
}
```

Stored properties

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

- If you create an instance of a structure, assign that instance to a constant and change it, there is an error:

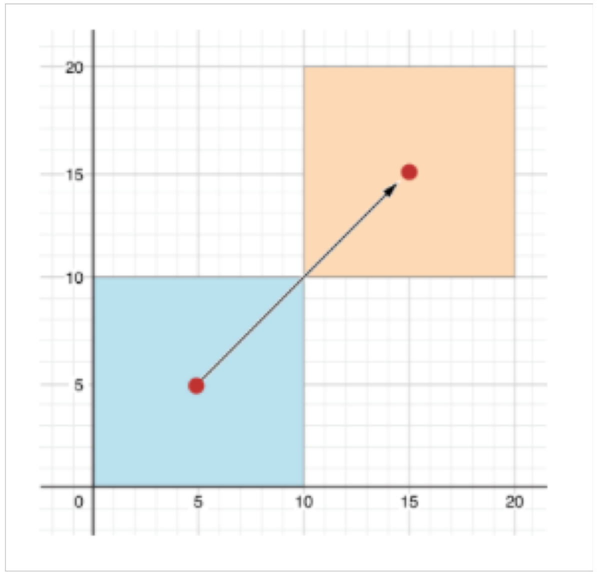
```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even though firstValue is a variable property
```

- A lazy stored property is a property whose initial value is not calculated until the first time it is used

```
class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
}
```

Computed properties

```swift
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
    size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// prints "square.origin is now at (10.0, 10.0)"
```



Read-only computed properties

- A computed property with a getter but no setter is known as a read-only computed property
  - We can remove the get keyword and its braces

```swift
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// prints "the volume of fourByFiveByTwo is 40.0"
```

Property Observers

- They observe and respond to changes in property values
  - They are called every time the property’s setter is called
  - cannot be added to lazy properties
- One can implement
  - willSet - called just before the value is stored
  - didSet - called immediately after the new value is stored

```
class StepCounter {
  var tSteps: Int = 0 {
    willSet(nSteps) {
      print("About to set tSteps to \(nSteps)")
    }
    didSet {
      if tSteps > oldValue  {
        print("Added \(tSteps - oldValue) steps")
      }
    }
  }
}
let stepCounter = StepCounter()
stepCounter.tSteps = 200
// About to set tSteps to 200
// Added 200 steps
stepCounter.tSteps = 360
// About to set tSteps to 360
// Added 160 steps
stepCounter.tSteps = 896
// About to set tSteps to 896
// Added 536 steps
```

Type Properties (static)

- They belong to the type and not to a specific instance!
- For computed type properties for class types, we can use the class keyword instead to allow subclasses to override the superclass's implementation

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}

class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

Methods

- Methods are functions that are associated with a particular type
  - Classes, structures, and enumerations can all define methods
  - Support for instance methods and type methods

```swift
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
    func incrementBy(_ amount: Int) {
        self.count += amount
    }
    func reset() {
        count = 0
    }
}

let counter = Counter()
counter.increment()
counter.incrementBy(5)
counter.reset()
```

Modifying Value Types from Within Instance Methods

- Structures and enumerations are value types
- By default, the properties of a value type cannot be modified from within its instance methods
  - Unless the method is identified as mutating
  - The method assigns a completely new instance to its implicit self property

```swift
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveByX(x: 2.0, y: 3.0)
print("The point is now at (\(somePoint.x), \(somePoint.y))")
// prints "The point is now at (3.0, 4.0)"
```

Type Methods

- We indicate type methods for classes by writing keyword **class** before func
- We indicate type methods for structures by writing keyword **static** func
- Self becomes a reference to the type itself, and not to a particular instance
- Calls to these methods are made directly to the type and not to an instance

Subscripts

- Subscripts are shortcuts for accessing the member elements of a collection, list or sequence
  - Access through indexes and not specific getters and setters
  - Multiple subscripts for a type can coexist, and the correct subscript is chosen based on the type of index value passed
  - subscripts can have multiple dimensions (not just a single index parameter)
- They can be used to query an instance
  - introduced by keyword subscript

```
subscript(index: Int) -> Int {
    get {
        // return an appropriate value
    }
    set(newValue) {
        // perform a setting action here
    }
}
```

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTTable = TimesTable(multiplier: 3)
print("six x three is \(threeTTable[6])")
// prints "six x three is 18"
```

Example

```
struct Matrix {
    let rows: Int, columns: Int
    var grid: [Double]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(repeating: 0.0, count: rows * columns)
    }
    func indexIsValid(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
       get {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            return grid[(row * columns) + column]
       }
       set {
            assert(indexIsValid(row: row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
       }
    }
}

var matrix = Matrix(rows: 2, columns: 2)

matrix[0, 1] = 1.5
matrix[1, 0] = 3.2
```

Inheritance

- Swift classes do NOT inherit from a universal base class
  - Every class that does not inherit from another class becomes a base class

```
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }
    func makeNoise() {
        // do nothing - an arbitrary vehicle doesn't necessarily make a noise
    }
}

class Bicycle: Vehicle {
    var hasBasket = false
}

class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}
```

Overriding

- Overriding
  - The method should be introduced by keyword **override**
  - can be prevented using keyword **final**
- Accessing the superclass is done using keyword **super**
  - An overridden method named someMethod can call the superclass version of someMethod by calling super.someMethod() within the overriding method implementation
  - An overridden property called someProperty can access the superclass version of someProperty as super.someProperty within the overriding getter or setter implementation
  - An overridden subscript for someIndex can access the superclass version of the same subscript as super[someIndex] from within the overriding subscript implementation

Examples

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}

class Car: Vehicle {
    var gear = 1
    override var description: String {
        return super.description + " in gear \(gear)"
    }
}
```

Initialization

- Initialization involves setting an initial value for each stored property on the instance and performing any other setup or initialization that is required before the new instance is ready for use
  - Swift initializers do not return a value
- Instances of class types can also implement a deinitializer, which performs any custom cleanup just before an instance of that class is deallocated

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
print("The default temperature is \(f.temperature)° Fahrenheit")
// prints "The default temperature is 32.0° Fahrenheit"
```

Inizialization

- Optional properties are automatically set to nil during initialization
- Constant properties can be modified at any point during initialization
  - As long as they are set to a definite value by the time initialization finishes
- Default initializers are provided when no other initializers are provided by the programmer
  - Only for base classes
  - The default initializer simply creates a new instance with all of its properties set to their default values
- Structures automatically receive memberwise initializers, if they do not provide their own
  - Initial values for the properties of the new instance can be passed to the memberwise initializer by name

Example

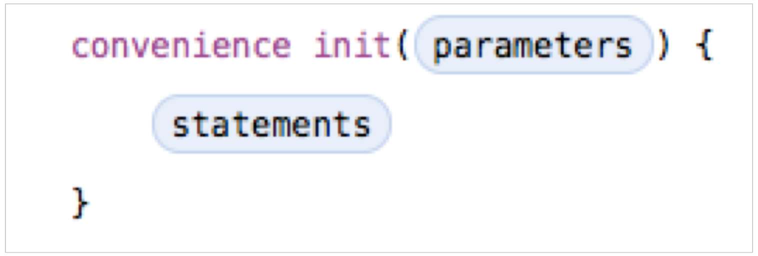
```swift
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

Initializer Delegation

- Initializers can call other initializers
  - To avoid duplicate code
- The rules are different for value types and class types
  - Value types (structures) do not allow inheritance
    - They can only delegate to initializers that they provide themselves
    - If we define a custom initializer for a value type, we will no longer have access to the default
  - Classes must ensure that all the stored properties they inherit are assigned a suitable value
    - This is achieved through **designated** initializers and **convenience** initializers
- Swift subclasses do not inherit their superclass initializers by default

Initializers

- A designated, primary, initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain
  - Every class must have at least one designated initializer, and often it is only one
- Convenience initializers are secondary, supporting initializers for a class
  - Create convenience initializers whenever a shortcut to a common initialization pattern will save time or make initialization of the class clearer in intent



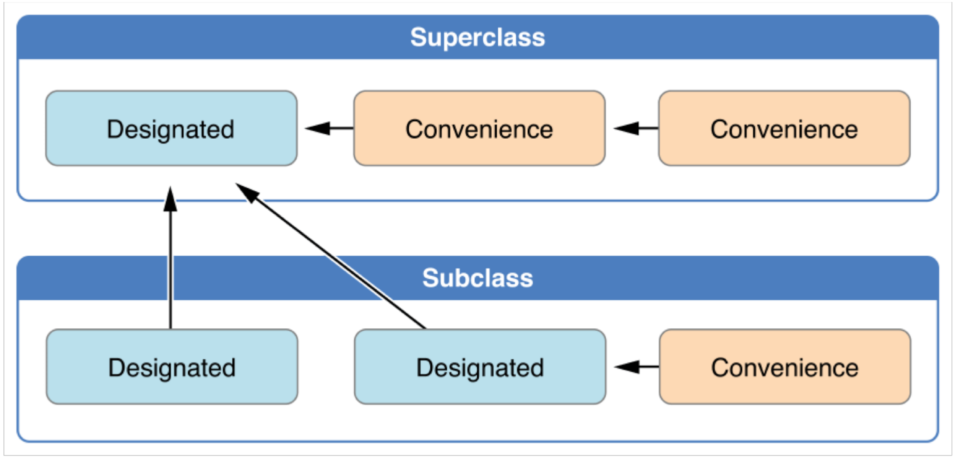
```
convenience init( parameters ) {
    statements
}
```

Rules for Delegation Calls

- **Rule 1:** A designated initializer must call a designated initializer from its immediate superclass
- **Rule 2:** A convenience initializer must call another initializer from the same class
- **Rule 3:** A convenience initializer must ultimately call a designated initializer

Designated initializers must always delegate up

Convenience initializers must always delegate across



Two-phase Initialization

- Each stored property is assigned an initial value by the class that introduced it
- Each class is given the opportunity to customize its stored properties further before the new instance is considered ready for use

- This is to prevent property values
  - from being accessed before they are initialized
  - from being set to a different value by another initializer unexpectedly

Compiler performs 4 safety checks

- A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer
- A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property
  - If it doesn't, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization
- A convenience initializer must delegate to another initializer before assigning a value to any property (including properties defined by the same class)
  - If it doesn't, the new value the convenience initializer assigns will be overwritten by its own class's designated initializer
- An initializer cannot call any instance methods, read the values of any instance properties, or refer to self as a value until after the first phase of initialization is complete

Example

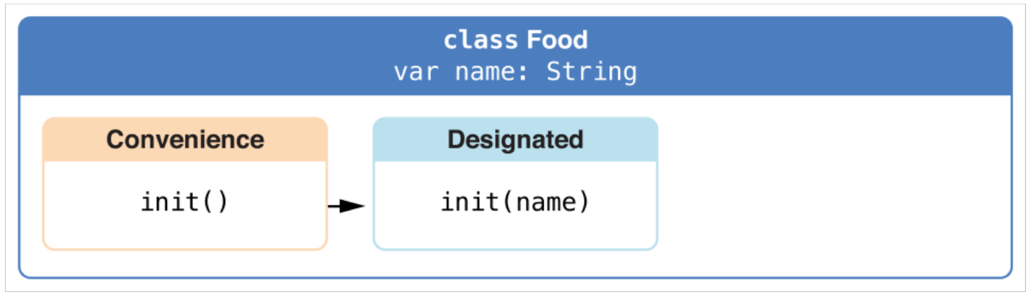
```swift
class Vehicle {
    var numberOfWheels = 0
    var description: String {
        return "\(numberOfWheels) wheel(s)"
    }
}

let vehicle = Vehicle()
print("Vehicle: \(vehicle.description)")
// Vehicle: 0 wheel(s)

class Bicycle: Vehicle {
    override init() {
        super.init()
        numberOfWheels = 2
    }
}

let bicycle = Bicycle()
print("Bicycle: \(bicycle.description)")
// Bicycle: 2 wheel(s)
```

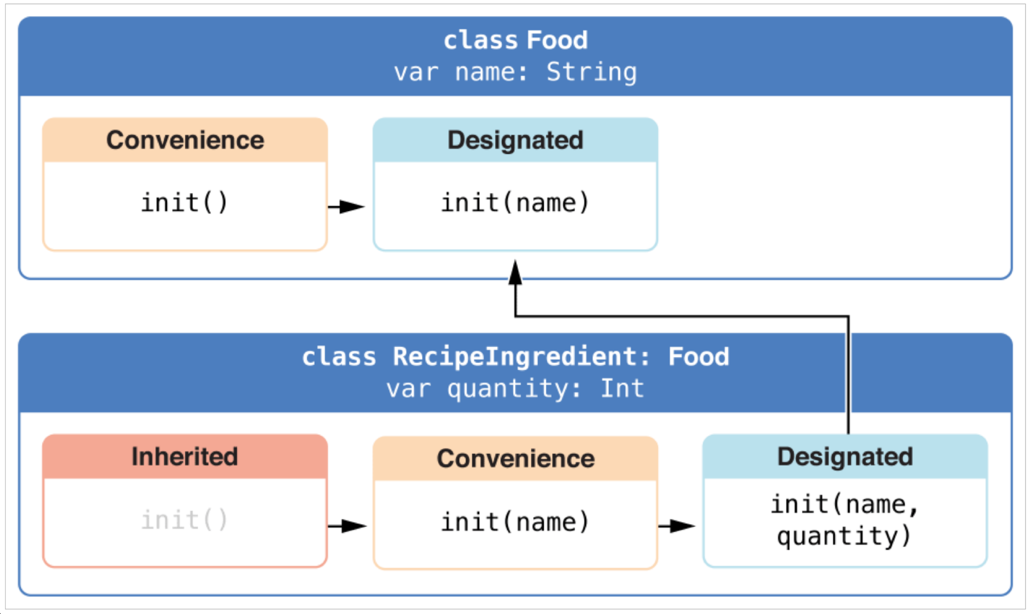
Automatic Initializer Inheritance

- If we provide default values for any new properties introduced in a subclass
  - **Rule 1:** If the subclass doesn’t define any designated initializers, it automatically inherits all of its superclass designated initializers
  - **Rule 2** If the subclass provides an implementation of all of its superclass designated initializers, then it automatically inherits all of the superclass convenience initializers
- These rules apply even if your subclass adds further convenience initializers

Example

```
class Food {
    var name: String
    init(name: String) {
       self.name = name
   }
    convenience init() {
        self.init(name: "[Unnamed]")
   }
}

let namedMeat = Food(name: "Bacon")
// namedMeat's name is "Bacon"

let mysteryMeat = Food()
// mysteryMeat's name is "[Unnamed]"

class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
   }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
   }
}

let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```





Failable Initializers

- Initialization may fail because of invalid initialization parameter values, absence of a required external resource, or other reasons
  - We write a failable initializer by placing a question mark after the init keyword (init?)
  - We write return nil to indicate a point at which initialization failure can be triggered

```
struct Animal {
    let species: String
    init?(species: String) {
        if species.isEmpty { return nil }
        self.species = species
    }
}

let anonymousCreature = Animal(species: "")
// anonymousCreature is of type Animal?, not Animal

if anonymousCreature == nil {
    print("The anonymous creature could not be initialized")
}
```

Required Initializers

- The required modifier before the definition of a class initializer indicates that every subclass of the class must implement that initializer
  - The required modifier before every subclass implementation of a required initializer indicates that the initializer requirement applies to further subclasses in the chain
    - We do not use the override modifier when overriding a required designated initializer

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}

class SomeSubclass: SomeClass {
    required init() {
        // subclass implementation of the required initializer goes here
    }
}
```

Deinitialization

- Called immediately before a class instance is deallocated
  - use the deinit keyword
  - at most 1 deinitializer per class
  - no params and written without parenthesis
- Deinitializers are called automatically
  - Superclass deinitializers are inherited
  - Superclass deinitializers are automatically called after deinitialization of the subclass
- Swift automatically deallocates instances when they are no longer needed
  - achieved through Automatic Reference Counting (ARC)

Extensions

- Extensions add new functionality to an existing class, structure, or enumeration type
- This includes the ability to extend types for which you do not have access to the original source code
  - Add computed instance properties and computed type properties
  - Define instance methods and type methods
  - Provide new initializers
  - Define subscripts
  - Define and use new nested types
  - Make an existing type conform to a protocol
- Extensions can add new functionality to a type, but they cannot override existing functionality

Example

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}

let oneInch = 25.4.mm
print("One inch is \(oneInch) meters")
// prints "One inch is 0.0254 meters"

let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")
// prints "Three feet is 0.914399970739201 meters"

let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
// prints "A marathon is 42195.0 meters long"
```

Protocols

- A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality
  - Any type that satisfies the requirements of a protocol is said to conform to that protocol
  - The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements
- Protocols can be composed

```
protocol SomeProtocol {
    // protocol definition goes here
}

struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}

class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol {
    // class definition goes here
}
```

Property Requirements

- A protocol can require any conforming type to provide an instance property or type property with a particular name and type
- The protocol does NOT specify whether the property should be a stored property or a computed property
  - it only specifies the required property name and type
  - The protocol also specifies whether each property must be gettable or gettable and settable
  - Property requirements are always declared as variable properties, prefixed with the var keyword

```
protocol SomeProtocol {
    var mustBeSettable: Int { get set }
    var doesNotNeedToBeSettable: Int { get }
}

protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

Example

```swift
protocol FullyNamed {
    var fullName: String { get }
}

struct Person: FullyNamed {
    var fullName: String
}
let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"


class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
        return (prefix != nil ? prefix! + " " : "") + name
    }
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName is "USS Enterprise"
```

Method Requirements

```
protocol SomeProtocol {
    static func someTypeMethod()
}

protocol RandomNumberGenerator {
    func random() -> Double
}

class LinearCongruentialGenerator: RandomNumberGenerator {
   var lastRandom = 42.0
   let m = 139968.0
   let a = 3877.0
   let c = 29573.0
    func random() -> Double {
        lastRandom = ((lastRandom * a + c).truncatingRemainder(dividingBy:m))
        return lastRandom / m
   }
}
let generator = LinearCongruentialGenerator()
print("Here's a random number: \(generator.random())")
// Prints "Here's a random number: 0.37464991998171"
print("And another one: \(generator.random())")
// Prints "And another one: 0.729023776863283"
```

Initializer Requirements

```
protocol SomeProtocol {
    init()
}

class SomeClass: SomeProtocol {
    required init() {
        // initializer implementation goes here
    }
}

class SomeSuperClass {
    init() {
        // initializer implementation goes here
    }
}

class SomeSubClass: SomeSuperClass, SomeProtocol {
    // "required" from SomeProtocol conformance; "override" from SomeSuperClass
    required override init() {
        // initializer implementation goes here
    }
}
```

Protocols and Extensions

```
protocol TextRepresentable {
    var textualDescription: String { get }
}

class Dice {
    let sides: Int
    let generator: RandomNumberGenerator
    init(sides: Int, generator: RandomNumberGenerator) {
        self.sides = sides
        self.generator = generator
    }
    func roll() -> Int {
        return Int(generator.random() * Double(sides)) + 1
    }
}

extension Dice: TextRepresentable {
    var textualDescription: String {
        return "A \(sides)-sided dice"
    }
}

var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
print(d6.textualDescription)
// prints "A 6-sided dice"
```

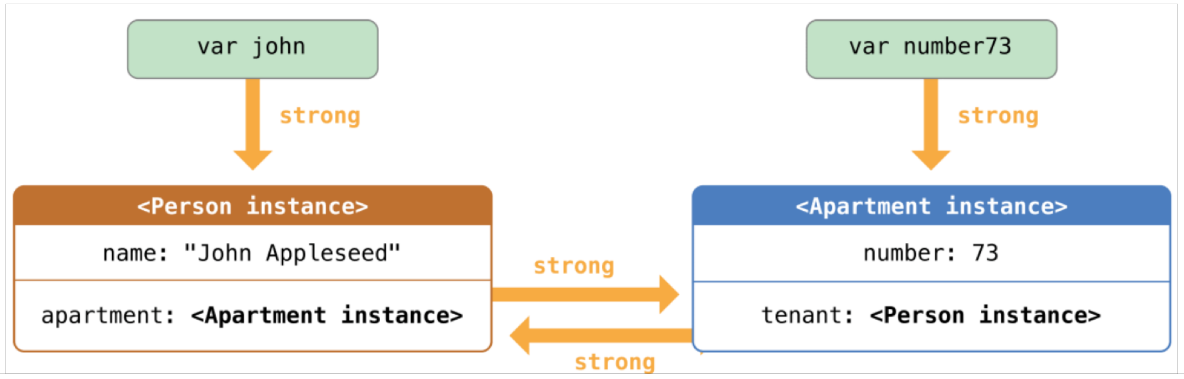
Automatic Reference Counting (ARC)

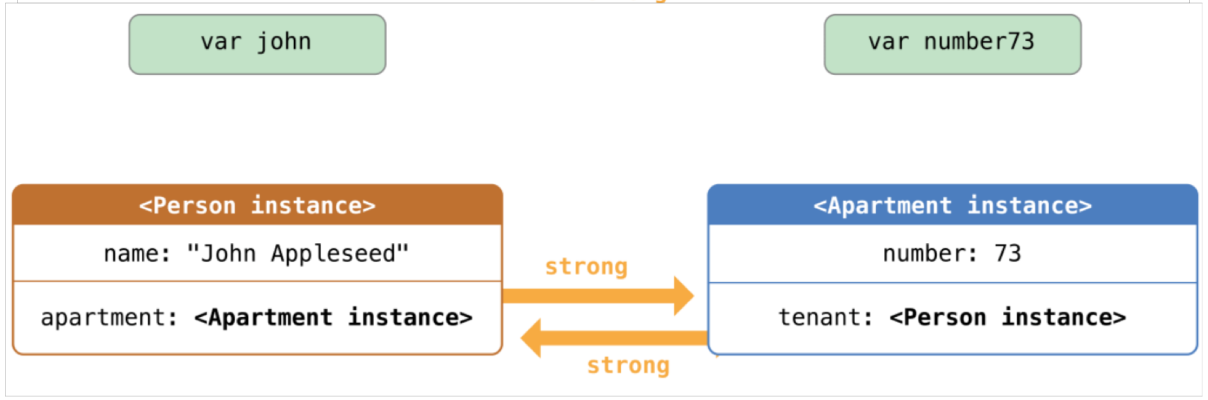
- Swift tracks and manages memory usage
  - In most cases, this means that memory management “just works” in Swift
  - ARC automatically frees up the memory used by class instances when those instances are no longer needed
  - In a few cases ARC requires more information about the relationships between parts of our code to make sure that instances don’t disappear while they are still needed
- ARC tracks how many properties, constants, and variables are currently referring to each class instance
  - ARC will not deallocate an instance as long as at least one active reference to that instance still exists
  - The reference is called a “strong“ reference because it keeps a firm hold on that instance, and does not allow it to be deallocated for as long as that strong reference remains

```swift
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}

var reference1: Person?
var reference2: Person?
var reference3: Person?

reference1 = Person(name: "John Appleseed")
// prints "John Appleseed is being initialized"

reference2 = reference1
reference3 = reference1

reference1 = nil
reference2 = nil

reference3 = nil
// prints "John Appleseed is being deinitialized"
```

Strong Reference Cycles

- Is it possible to never reach a point that we have zero references?
  - It can happen if two class instances hold a strong reference to each other
  - We resolve strong reference cycles by defining some of the relationships between classes as weak references

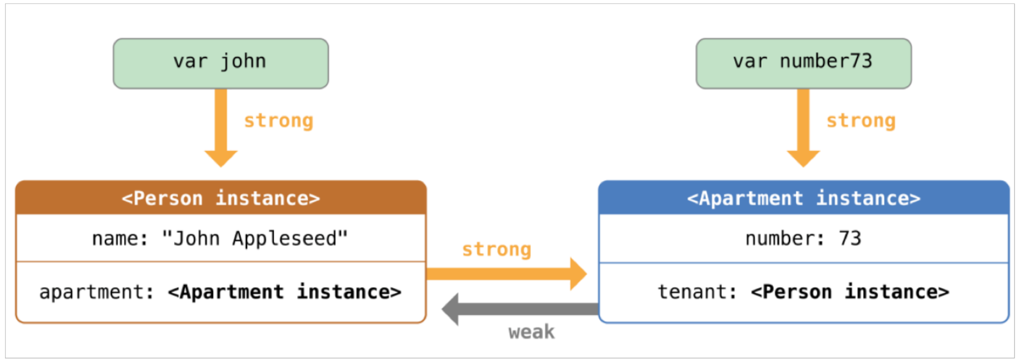




Weak References

- Use a weak reference whenever it is valid for that reference to become nil at some point during its lifetime
  - It is appropriate for an apartment to be able to have “no tenant” at some point in its lifetime
  - Because weak references are allowed to have “no value”, every weak reference is an optional type
  - ARC automatically sets a weak reference to nil when the instance that it refers to is deallocated

```
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}
```



Unowned References

- Conversely, use an unowned reference when you know that the reference will never be nil once it has been set during initialization
  - An unowned reference is assumed to always have a value
  - It is always defined as a non-optional type
  - We don’t need to unwrap the unowned reference each time it is used

Access Control

- Swift's access control model is based on the concept of modules and source files
  - A module is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift's import keyword
  - A source file is a single Swift source code file within a module (in effect, a single file within an app or framework)
- Swift supports three access levels
  - Public access enables entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module
  - Internal access enables entities to be used within any source file from their defining module, but not in any source file outside of that module
  - Private access restricts the use of an entity to its own defining source file

Guiding Principals

- No entity can be defined in terms of another entity that has a lower (more restrictive) access level
  - For example, a function cannot have a higher access level than its parameter types and return type, because the function could be used in situations where its constituent types are not available to the surrounding code
- All entities in your code have a default access level of internal if you do not specify an explicit access level yourself
  - Define the access level for an entity by placing one of the public, internal, or private modifiers before the entity
- Some rules
  - The access level for a tuple type is the most restrictive access level of all types used in that tuple
  - The access level for a function type is calculated as the most restrictive access level of the function's parameter types and return type
  - A subclass cannot have a higher access level than its superclass, but an override can make an inherited class member more accessible than its superclass version

Grand Central Dispatch (GCD)

- A queue is a block of code that can be executed synchronously or asynchronously, either on the main or on a background thread
- Once a queue is created, the operating system manages it
  - Queues follow the FIFO pattern (First In, First Out)
  - A queue can be either serial or concurrent
- A work item is a block of code that is either written along with the queue creation, or it gets assigned to a queue and it can be used more than once (reused)
  - The execution of work items in a queue also follows the FIFO pattern
  - This execution can be synchronous or asynchronous.

Example

```swift
import Dispatch

let main = DispatchQueue.main
let background = DispatchQueue.global()

func doSyncWork() {
    background.sync { for _ in 1...3 { print("Light") } }
    for _ in 1...3 { print("Heavy") } }

func doAsyncWork() {
    background.async { for _ in 1...3 { print("Light") } }
    for _ in 1...3 { print("Heavy") } }
```