Swift

- **New language features**
  - type inference (strongly typed language, no need to specify variable type in most cases)
  - super switch statement
  - closures (functional programming support for lambda expression)
  - tuples (collection of values of different types)
  - optionals (new type for variables that might not have a value)
  - Array, String and Dictionary types (value types, not referenced, support for bounds checking)
  - Enhanced struct and enum type (can have many class-like features)
  - functions can return multiple values (tuples)
  - operator overloading (including definition of new operators)
  - generics (write code with placeholders for data type translation at compile time)
  - integer overflow checking (arithmetic overflow result in runtime error)
  - String interpolation (build strings by inserting expression in literals into placeholders)
  - Nested types and functions!

Safer Language

- **{} required around body of every control statement**
  - ensure you do not accidentally omit braces around multi-statement bodies
- **No pointers unlike Objective-C, C and C++**
- **The assignment operator (=) does not return a value**
  - compilation error occurs if assignment used in a condition rather ==
- **Semicolons are optional except for multiple statements on the same line**
- **Parentheses around conditions in control statements are optional**
- **Variables and constants must be initialised before they are used**
  - in their definitions, or
  - via initializer methods in type definitions
- **Integer calculations are checked for overflow**
  - runtime error occurs if a calculation results in overflow

Safer Language

- **Strongly typed language**
  - No implicit conversions between numeric types
  - Variable and constant types can be inferred in most cases
- **Array indices are bounds checked at execution time**
  - runtime error if you try to access an element outside an Array's bounds
- **Automatic memory management**

  →eliminates most memory leaks

  →use automatic reference counting (no garbage collection, way better!)

  →possibility to keep references to objects that are no longer used

  →weak references to avoid circular references between objects preventing memory from being reclaimed

Swift 4

- **Open Source with initial port for MacOS, iOS, Linux**
- **Source code for compiler and standard library**
- **Contribution from community encouraged**
- **New Language features:**
  - error handling model (try, throw, catch, ErrorType)
  - improved and simplified optional binding patterns
  - clean-up actions
  - protocol extensions

Variables

```
var languageName: String = "Swift"
var version: Double = 4.2
var released: Int = 2018
var introduced: Int = 2014
var isAwesome: Bool = true
```

Variables and Constants

```
let languageName: String = "Swift"
var version: Double = 4.2
var released: Int = 2018
let introduced: Int = 2014
let isAwesome: Bool = true
```

Type Inference

```
let languageName = "Swift"  // inferred as String
var version = 4.2           // inferred as Double
var released = 2018         // inferred as Int
let introduced = 2014       // inferred as Int
let isAwesome = true        // inferred as Bool
```

Type Inference and Unicode Names

```
let languageName = "Swift"
var version = 4.2
var released = 2018
let introduced = 2014
let isAwesome = true
let π = 3.1415927
let 🐶🐮 = "dogcow"
```

Swift – Optional Type

- Swift introduces a new type called Optional
- An Optional is conceptually just an generic enum type

```
enum Optional<Wrapped_Type> {
   case none
   case some(Wrapped_Type)
}
```

- Examples:

```
let x: String? = nil
// is equivalent to
let x = Optional<String>.none

let x: String? = "Hello UCD"
// is equivalent to
let x = Optional<String>.some("Hello UCD")

var y = x!
// is equivalent to
switch x {
   case some(let value): y = value
   case none: // raise an exception
}
```

Swift – Array

- Array:

```
var a = Array<String>()
// is equivalent
var a = [String]()


let cars = ["VW", "Renault", "Ford", "Audi"]

cars.append("Seat") // won't compile, cars is immutable (because of let)

let car = cars[5] // crash (array index out of bounds)


// enumerating an Array
for car in cars {
    print("\(car)")
}
```

Swift – Dictionary

- Dictionary

```
var vehicles = Dictionary<String, Int>()
// is equivalent to ...
var vehicles = [String:Int]()

vehicles = ["Car": 4, "Tricyle": 3, "Bike": 2]
let wheels = vehicles["Truck"] // wheels is an Int? (would be nil)

// use a tuple with for-in to enumerate a Dictionary
for (key, value) in vehicles {
    print("\(key) = \(value)")
}
```

Swift – Range

- A Range in Swift is just two end points of a sensible type
- Range is generic i.e Range<T>, pseudo-representation :

```
struct Range<T> {
    var startIndex: T
    var endIndex: T
}
```

- An Array's range would be a Range<Int> (since Arrays are indexed by Int)
- Warning: A String subrange is not Range<Int> (it is Range<Index> ... cf docs)

- There is special syntax for specifying a Range:
  - either ... (inclusive)
  - or ..< (open-ended)

```
let array = ["a", "b", "c", "d"]
let subArray1 = array[2...3] // subArray1 will be ["c", "d"]
let subArray2 = array[2..<3] // subArray2 will be ["c"]
for i in 27...104 { } // Range can be enumerated, like Array, String, Dictionary
```

Swift – Other Classes

- **NSObject**
  - Base class for all Objective-C classes
  - Some advanced features will require you to subclass from NSObject (and it can't hurt to do so)
- **NSNumber**
  - Generic number-holding class

```
let n = NSNumber(3.1416)

let intVersion = n.intValue // also doubleValue, boolValue, etc...
```

- **NSDate**
  - Used to find out the date and time right now or to store past or future dates.
  - See also NSCalendar, NSDateFormatter, NSDateComponents
  - If you are displaying a date in your UI, there are localization ramifications, so check docs
- **NSData**
  - A "bag o' bits".
  - Used to save/restore/transmit raw data throughout the iOS SDK

Swift – Classes, Struct & Enum

- **These are the 3 fundamental building blocks of data structures in Swift:**
  - Classes
  - Structures
  - Enumerations
- **Similarities between data structures in Swift:**
  - Declaration syntax

```
class CalcModel {
}

struct Polygon {
}

enum Operation {
}
```

Swift – Classes, Struct & Enum

- **These are the 3 fundamental building blocks of data structures in Swift:**
  - Classes
  - Structures
  - Enumerations
- **Similarities between data structures in Swift:**
  - Declaration syntax
  - Properties and functions

```
func aFunction(argument: Type) -> ReturnValue {
    // ...
}

var storedProperty = <initial value> // (not enum)

var computedProperty: Type {
    get {}
    set {}
}
```

Swift – Classes, Struct & Enum

- **These are the 3 fundamental building blocks of data structures in Swift:**
  - Classes
  - Structures
  - Enumerations
- **Similarities between data structures in Swift:**
  - Declaration syntax
  - Properties and functions
  - Initializers (except for enum)

```
init(argument1: Type, argument2: Type, ...) {
    // ...
}
```

Swift – Classes, Struct & Enum

- **These are the 3 fundamental building blocks of data structures in Swift:**
  - Classes
  - Structures
  - Enumerations
- **Similarities between data structures in Swift:**
  - Declaration syntax
  - Properties and functions
  - Initializers (except for enum)
- **Differences:**
  - Inheritance (class only)
  - Introspection and casting (class only)
  - Value type (struct, enum) versus Reference type (class)

Swift – Value versus Reference

- **Value (struct and enum)**
  - Copied when passed as an argument to a function
  - Copied when assigned to a different variable
  - Immutable if assigned to a variable with let
  - Remember that function parameters are, by default, constants
  - You can put the keyword var on an parameter, and it will be mutable, but it's still a copy
  - You must note any func that can mutate a struct/enum with the keyword mutating
- **Reference (class)**
  - Stored in the heap and reference counted (automatically)
  - Constant pointers to a class (let) still can mutate by calling methods and changing properties
  - When passed as an argument, does not make a copy (just passing a pointer to same instance)
- **Choosing which to use?**
  - Usually you will choose class over struct.
  - struct tends to be more for fundamental types
  - Use of enum is situational (any time you have a type of data with discrete values)

Swift – Methods

- **Obviously you can override methods/properties in your superclass**
  - Precede your func or var with the keyword override
  - A method can be marked final which will prevent subclasses from being able to override
  - Classes can also be marked final
- **Both types and instances can have methods/properties**
  - For this example, lets consider the struct Double

```
var d: Double = -3.1416
if d.isSignMinus {
    d = Double.abs(d)
}
```

  - isSignMinus is an instance property of a Double (you send it to a particular Double)
  - abs is a type method of Double (you send it to the type itself, not to a particular Double)
  - You declare a type method or property with a static prefix (or class in a class) ...

```
static func abs(d: Double) -> Double
```

Swift – Methods

- **Parameters Names**
  - All parameters to all functions have an internal name and an external name
  - The internal name is the name of the local variable you use inside the method

```
func foo(external internal: Int) {
    let local = internal
}

func boo() {
    let result = foo(external: 123)
}
```

Swift – Methods

- **Parameters Names**
  - All parameters to all functions have an internal name and an external name
  - The internal name is the name of the local variable you use inside the method
  - The external name is what callers will use to call the method

```
func foo(external internal: Int) {
    let local = internal
}

func boo() {
    let result = foo(external: 123)
}
```

Swift – Methods

- **Parameters Names**
  - All parameters to all functions have an internal name and an external name
  - The internal name is the name of the local variable you use inside the method
  - The external name is what callers will use to call the method
  - You can put _ if you don't want callers to use an external name at all for a given parameter

```
func foo(_ internal: Int) {
    let local = internal
}

func boo() {
    let result = foo(123)
}
```

Swift – Methods

- **Parameters Names**
  - All parameters to all functions have an internal name and an external name
  - The internal name is the name of the local variable you use inside the method
  - The external name is what callers will use to call the method
  - You can put _ if you don't want callers to use an external name at all for a given parameter
  - The internal name is, by default, the external name

```
func foo(first: Int, second: Double) {
    let local = first
}

func boo() {
    let result = foo(first: 2018, second: 3.1416)
}
```

Swift – Methods

- **Parameters Names**
  - All parameters to all functions have an internal name and an external name
  - The internal name is the name of the local variable you use inside the method
  - The external name is what callers will use to call the method
  - You can put _ if you don't want callers to use an external name at all for a given parameter
  - The internal name is, by default, the external name
  - Any parameter's external name can be changed

```
func foo(_ first: Int, externalSecond second: Double) {
    let local = first
}

func boo() {
    let result = foo(2018, externalSecond: 3.1416)
}
```

Swift – Properties

- **Property Observers**
  - You can observe changes to any property with willSet and didSet

```
var someStoredProperty: Int = 2018 {
    willSet {
        newValue // is the new value
    }
    didSet {
        oldValue // is the old value
    }
}

override var inheritedProperty {
    willSet {
        newValue // is the new value
    }
    didSet {
        oldValue // is the old value
    }
}
```

Swift – Properties

- **Property Observers**
  - You can observe changes to any property with willSet and didSet
  - One very common thing to do in an observer in a Controller is to update the user-interface
- **Lazy Initialization**
  - A lazy property does not get initialized until someone accesses it
  - You can allocate an object, execute a closure, or call a method if you want

```
lazy var calcModel = CalcModel() // nice if CalcModel used lots of resources

lazy var someProperty: Type = {
    // construct the value of someProperty here
    return <the constructed value>
}()

lazy var myProperty = self.initializeMyProperty()
```

  - This still satisfies the "you must initialize all of your properties" rule
  - Unfortunately, things initialized this way can't be constants (i.e. var ok, let not okay)
  - This can be used to get around some initialization dependency

Swift – Class Init

- **When is an init method needed?**
  - init methods are not so common because properties can have their defaults set using =
  - Or properties might be Optionals, in which case they start out nil
  - You can also initialize a property by executing a closure
  - Or use lazy instantiation
  - So you only need init when a value can't be set in any of these ways
- **You also get some "free" init methods**
  - If all properties in a base class (no superclass) have defaults, you get init() for free
  - If a struct has no initializers, it will get a default one with all properties as arguments

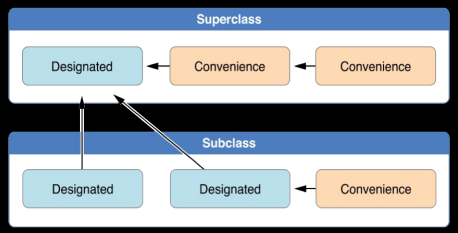
```
struct MyStruct {
    var year: Int = 2019
    var label: String = "🐖豬"
    init(year: Int, label: String) // comes for free
}
```

Swift – Class Init

- **What can you do inside an init?**
  - You can set any property's value, even those with default values
  - Constant properties (i.e. properties declared with let) can be set
  - You can call other init methods in your own class using self.init(<args>)
  - In a class, you can of course also call super.init(<args>)
  - But there are some rules for calling inits from inits in a class ...

Swift – Class Init

- **What are you required to do inside init?**
  - By the time any init is done, all properties must have values (optionals can have the value nil)
  - There are two types of inits in a class, convenience and designated (i.e. not convenience)
  - A designated init must (and can only) call a designated init that is in its immediate superclass
  - You must initialize all properties introduced by your class before calling a superclass's init
  - You must call a superclass's init before you assign a value to an inherited property
  - A convenience init must (and can only) call a designated init in its own class
  - A convenience init may call a designated init indirectly (through another convenience init)
  - A convenience init must call a designated init before it can set any property values
  - The calling of other inits must be complete before you can access properties or invoke methods



Swift – Class Init

- **Inheriting init**
  - If you do not implement any designated inits, you'll inherit all of your superclass's designated
  - If you override all of your superclass's designated inits, you'll inherit all its convenience inits
  - If you implement no inits, you'll inherit all of your superclass's inits
  - Any init inherited by these rules qualifies to satisfy any of the rules on the previous slide
- **Required init**
  - A class can mark one or more of its init methods as required
  - Any subclass must implement said init methods (though they can be inherited per above rules)

Swift – Class Init

- **Failable init**
  - If an init is declared with a ? after the word init, it returns an Optional

```
init?(arg1: Type1, ...) {
    // might return nil in here
}
```

  - These are rare.
  - Note: The documentation does not seem to properly show these inits!
  - But you'll be able to tell because the compiler will warn you about the type when you access it

```
let image = UIImage(named: "UCD_logo") // image is an Optional UIImage (i.e. UIImage?)
```

  - Usually we would use if-let for these cases…

```
if let image = UIImage(named: "UCD_logo") {
   // image was successfully created
} else {
   // couldn't create the image
}
```

Swift – Class Init

- **Creating Objects**
  - Usually you create an object by calling it's initializer via the type name …

```
let x = CalcModel()
let y = ComplexObject(arg1: 2017, arg2: "🐦💩", ...)
let z = [String]()
```

  - But sometimes you create objects by calling type methods in classes …

```
let button = UIButton(type: .System)
```

  - Or obviously sometimes other objects will create objects for you …

```
var myArray = ["1", "2", "3"]
let commaSeparatedArrayElements: String = myArray.joined(separator: ",")
```

Swift – AnyObject

- **Special "Type" (actually it's a Protocol)**
  - Used primarily for compatibility with existing Objective-C-based APIs
- **Where will you see it?**
  - As properties (either singularly or as an array of them), e.g. ...

```
var destinationViewController: AnyObject
var toolbarItems: [AnyObject]
```

  - ... or as arguments to functions ...

```
func prepare(for segue: UIStoryboardSegue, sender: AnyObject)
func addConstraints(constraints: [AnyObject])
func digitPressed(sender: AnyObject)
```

  - ... or even as return types from functions …

```
class func buttonWithType(buttonType: UIButtonType) -> AnyObject
```

Swift – AnyObject

- **How do we use AnyObject?**
  - We don't usually use it directly
  - Instead, we convert it to another, known type
- **How do we convert it?**
  - We need to create a new variable which is of a known object type (i.e. not AnyObject)
  - Then we assign this new variable to hold the thing that is AnyObject
  - Of course, that new variable has to be of a compatible type
  - If we try to force the AnyObject into something incompatible, crash!
  - But there are ways to check compatibility (either before forcing or while forcing)…

Swift – AnyObject

- **Casting AnyObject**
  - We "force" an AnyObject to be something else by "casting" it using the as keyword …
  - Let's use var destinationVC: AnyObject as an example …

```
let calcVC = destinationVC as CalcViewController
```

  - … this would crash if dvc was not, in fact, a CalculatorViewController (or subclass thereof)
  - To protect against a crash, we can use if let with as? …

```
if let calcVC = destinationViewController as? CalcViewController { ... }
```

  - … as? returns an Optional (calcVC = nil if dvc was not a CalculatorViewController)
  - Or we can check before we even try to do as with the is keyword …

```
if destinationVC is CalcViewController { ... }
```

Swift – AnyObject

- **Casting Arrays of AnyObject**
  - If you're dealing with an [AnyObject], you can cast the elements or the entire array …
  - Let's use var toolbarItems: [AnyObject] as an example ...

```
for item in toolbarItems { // item's type is AnyObject
    if let toolbarItem = item as? UIBarButtonItem {
        // do something with the toolbarItem (which will be a UIBarButtonItem here)
    }
}
```

  - ... or ...

```
for toolbarItem in toolbarItems as [UIBarButtonItem] { // better be so, else crash!
  // do something with the toolbarItem (which will be a UIBarButtonItem)
}
```

  - Can't do as? here because then it might be "for toolbarItem in nil" (makes no sense)

Swift – AnyObject

- **Another example …**
  - Remember when we wired up our Actions in our storyboard?
  - The default in the dialog that popped up was AnyObject.
  - We changed it to UIButton.
  - But what if we hadn't changed it to UIButton?
  - How would we have implemented appendDigit?

```
@IBAction func appendDigit(sender: AnyObject) {
    if let sendingButton = sender as? UIButton {
        let digit = sendingButton.currentTitle!
        // ...
    }
}
```

Swift – AnyObject

- **Yet another example …**
  - It is possible to create a button in code using a UIButton type method …

```
let button: AnyObject = UIButton.buttonWithType(UIButtonType.System)
```

  - The type of this button is AnyObject (for historical reasons only)
  - To use it, we'd have to cast button to UIButton

  - We can do this on the fly if we want …

```
let title = (button as UIButton).currentTitle

// this would crash if button was not, in fact, a UIButton
```

- **Swift3 also define type Any. Does not have to be an object. It is a little more general than AnyObject**

Swift – Casting

- **Casting is not just for AnyObject**
  - You can cast with as (or check with is) any object pointer that makes sense
  - For example ...

```
let vc: UIViewController = CalcViewController()
```

  - The type of vc is UIViewController (because we explicitly typed it to be)
  - And the assignment is legal because a CalcViewController is a UIViewController
  - But we can't say, for example, vc.pushOperand()

```
if let calcVC = vc as? CalcViewController {
    // in here we could say vc.pushOperand() if we wanted to
}
```

Swift – Protocol Functions & Array

- **Some Array<T> Protocol Methods**

```
var array: Array<Type>
// e.g. var array = [1, 3, 6]

array += [Type] // not += Type
array.first     // -> Type? note this is an optional
array.last      // -> Type? note this is an optional

array.append(Type)
array.insert(Type, at: Int)
// array.insert(2, at: 1) -> [1, 2, 3, 6]
array.insert(contentsOf: Array<Type>, at: Int)
// array.insert(contentsOf: [4, 5], at: 2) -> [1, 2, 3, 4, 5, 6]

array.remove(at: Int)
// array.remove(at: 3) -> [1, 2, 3, 5, 6]
array.removeSubrange(bounds: Range)
// array.removeSubrange(bounds: 0..<3) -> [5, 6]
array.replaceSubrange(bounds:Range, with: [Type])
// array.replaceSubrange(bounds 0...1, with: [8, 9, 10]) -> [8, 9 ,10]

array.sort((Type, Type) -> Bool) // array.sort { $0 < $1 }
```

Swift – Protocol Functions & Array

- **Some Array<T> Protocol Methods**
  - This one creates a new array with any "undesirables" filtered out
  - The function passed as the argument returns false if an element is undesirable

```
array.filter(includeElement: (Type) throws -> Bool)
// e.g. array.filter { $0 % 2 == 0 }
```

  - Create a new array by transforming each element to something different
  - The thing it is transformed to can be of a different type than what is in the Array

```
array.map(transform: (Type) throws -> T)
// e.g. array.map { "\($0)" }
```

  - Reduce an entire array to a single value

```
array.reduce(initial: T, combine: (T, Type) throws -> T)
// e.g.
let sum: Int = [1, 2, 3].reduce(0) { $0 + $1 } // sums up numbers in array
```

Strings & Characters

- **String: Series of Characters (as opposed to collection), unicode compliant**

```
let someString = "I appear to be a string"
// inferred to be of type String

let components = "~/COMP41550/Assignments".pathComponents
// ["~", "COMP41550", "Assignments"]
```

- **Different Than the Sum of Its Parts**

```
var letters: [Character] = ["c", "a", "f", "e"]
var string: String = String(letters)

print(letters.count) // 4
print(string) // cafe
print(string.characters.count) // 4

let acuteAccent: Character = "\u{0301}" // ´ COMBINING ACUTE ACCENT' (U+0301)

string.append(acuteAccent)
print(string.characters.count) // 4
print(string.characters.last!) // é

string.characters.contains("e") // false
string.characters.contains("´") // false
string.characters.contains("é") // true
```

Strings & Characters

- **Strings are not collections**
- **Strings provide views that conform to CollectionType**
  - characters is a collection of Character values, or extended grapheme clusters.
  - unicodeScalars is a collection of Unicode scalar values.
  - utf8 is a collection of UTF–8 code units.
  - utf16 is a collection of UTF–16 code units.

```
var letters: [Character] = ["c", "a", "f", "e"]
var string: String = String(letters)
let acuteAccent: Character = "\u{0301}" // ´ COMBINING ACUTE ACCENT' (U+0301)
string.append(acuteAccent)
```

```
string.characters

string.unicodeScalars

string.utf8

string.utf16
```

Character	c	a	f	é	
Unicode Scalar Value	c	a	f	e	´
UTF-8 Code Unit	99	97	102	101	204	129
UTF-16 Code Unit	99	97	102	101	769	

Strings & Characters

- **String.Index**
  - In Unicode, a given glyph might be represented by multiple Unicode characters e.g. accents
  - Therefore you can't index a String by Int (because it's a collection of characters, not glyphs)
  - So a lot of native Swift String functions take a String.Index to specify which glyph you want
  - You can get a String.Index by asking the string for its startIndex then advancing forward
  - You advance forward with the function (not method) advance(String.Index, Int)

```
var greetings = "Heo UCD!"
let index = greetings.startIndex.advancedBy(2) // index is a String.Index to 3rd glyph "o"
greetings.insertContentsOf("ll".characters, at: index) // greetings = "Hello UCD!"

let startIndex = greetings.startIndex.advancedBy(5)
let endIndex = greetings.endIndex.advancedBy(-1)
let subString =  greetings[startIndex..<endIndex] // subString = "UCD"
```

Strings

- **Other useful string functions**

```
greetings.hasPrefix(String) -> Bool
greetings.hasSuffix(String) -> Bool
greetings.endIndex -> Bool
Int(greetings) -> Int?
greetings.capitalizedString -> String
greetings.lowercaseString -> String
greetings.uppercaseString -> String
[String].joinWithSeparator(String) -> String
// ["U", "C", "D"].joinWithSeparator(",") = "U,C,D"
String.componentsSeparatedByString(String) -> [String]
// "U,C,D".componentsSeparatedByString(",") -> ["U", "C", "D"]
```

Character

```
for character in "mouse".characters {
    print(character)
}

print("character count:","mouse".characters.count)

print("scalar count:","mouse".utf8.count)
```

m
o
u
s
e
character count: 5
scalar count: 5

Character

```
for character in "🐭🐭🐭🐭🐭".characters {
    print(character)
}

print("character count:","🐭🐭🐭🐭🐭".characters.count)

print("scalar count:","🐭🐭🐭🐭🐭".utf8.count)
```

🐭
🐭
🐭
🐭
🐭
character count: 5
scalar count: 20

Combining Strings and Characters

```
let dog: Character = "🐶"
let cow: Character = "🐮"

let dogCow = String(dog) + String(cow) // dogCow is "🐶🐮"

let instruction = "Beware of the " + String(dog)
// instruction is "Beware of the 🐶"
```

String Interpolation

```
let a = 3, b = 5

// "3 times 5 is 15"

let mathResult = "\(a) times \(b) is \(a * b)"

// "3 times 5 is 15"
```

String Interpolation

```
let a = 7, b = 4

// "7 times 4 is 28"

let mathResult = "\(a) times \(b) is \(a * b)"

// "7 times 4 is 28"
```

String Mutability

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

let constantString = "Highlander"
constantString += " and another Highlander"
// error – constantString cannot be changed
```

Type Conversion

- **Conversion between types with init()**
  - A sort of "hidden" way to convert between types is to create a new object by converting

```
let d: Double = 3.14
let f: Float = 3.14
let x = Int(d) // x = 3
let xd = Double(x)
let cgf = CGFloat(d)
let a = [Character]("UCD".characters) // a = ["U","C","D"] ie Array of Character
let s = String(a) // s = "UCD" // note a is an Array of Character
let si = String(x) // si = "3", no floats
let sf = "\(f)" // sf = "3.14", but you can use string interpolation instead
```

Swift – Extensions

- **Extensions:**
  - You can add methods and properties to a class
  - Possible even if you don't have the source
  - Not possible to re-implement already existing methods or properties (only add new ones)
  - Added properties can not have storage associated with them.
- **This feature is easily abused**
  - Should be used to add clarity to readability not obfuscation!
  - Don't use it as a substitute for good object-oriented design technique.
  - Best used (at least for beginners) for very small, well-contained helper functions.
  - Can actually be used well to organize code but requires architectural commitment.
  - When in doubt (for now), don't do it.

Swift 2 – Error Handling

- **Errors are represented by values whose type conform to the Error protocol**

```
enum EngineErrors: Error {
    case LowPetrol
    case LowOil
    case LowBattery
}
```

- **Function can throw an error**

```
let reserve = 5, discharged = 5.0
var petrol = 20, oilStatus = true, battery = 12.0

// Function throwing error
func engineCheck() throws {
    guard petrol > reserve else {
        throw EngineErrors.LowPetrol
    }

    guard oilStatus else {
        throw EngineErrors.LowOil
    }

    guard battery > discharged else {
        throw EngineErrors.LowBattery
    }
}
```

Swift 2 – Error Handling

- **Handling errors**

```
func startEngine() {
    do {
        try engineCheck()
        print("Engine started")
    } catch EngineErrors.LowPetrol {
        print("Refuel!")
    } catch EngineErrors.LowOil {
        print("Check Oil!")
    } catch EngineErrors.LowBattery {
        print("Low Battery!")
    } catch {
        // Default
        print("Unknown reason!")
    }
}
```

- **Side effect to new support for error handling: do-while replaced by repeat-while**

Swift 2 – Optional Binding

- **Improved handling of optional binding**
- **Possibility to test multiple binding at once**

```
func filterByAge(name: String?, age: Int?)  {
    guard let name = name, age = age where name.characters.count > 0 && age >= 18 else {
        print("not allowed")
        return
    }
    print("\(name) is allowed to vote")
}


filterByAge(nil, age: 21)           // Output?
filterByAge("AppleSeed", age: 7)    // Output?
filterByAge("", age: 21)            // Output?
filterByAge("AppleSeed", age: 21)   // Output?
```

Swift 2 – Cleanup Actions

- **defer statement defers execution until the current scope is exited**

```
func processFile(filename: String?) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            print("\(line)")
            // Work with the file
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

Swift 2 – Protocol Extensions

- **Class, enum, struct and now protocols can be extended without the need to subclass**

```
extension CustomStringConvertible {
    var loudDescription: String {
        return "\(self.description.uppercaseString)"
    }
}

let greetings = ["Hello", "UCD"]

print("\(greetings.description)")
print("\(greetings.loudDescription)")
```

- **Powerful feature**

```
let numbers = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
// Swift 1
let list = find(filter(map(numbers, { $0 * 2}), { $0 % 3 == 0 }), 42))

// Swift 2
let list = numbers.map { $0 * 2 }.filter { $0 % 3 == 0 }.indexOf(42)
print("\(list)")          // Output?
```

Swift – Debugging, Assertions

- **Intentionally crash your program if some condition is not true (and give a message)**

```
assert(condition:  @autoclosure () -> Bool, message: String)
```

– The function argument is an "autoclosure" however, so you don't need the { }

```
assert(validation() != nil, "validation returned nil")
```

– Will crash if validation() returns nil (because we are asserting that validation() does not)

– The validation() != nil part could be any code you want

- **When building for release (to the AppStore or whatever), asserts are ignored completely**

Xcode

Xcode Key Features

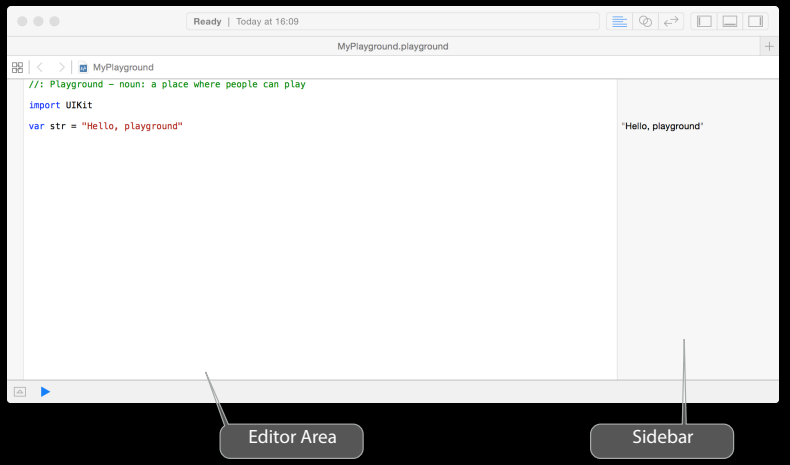
- **Playground**
  - Xcode window for inputting Swift code
  - executes as you type, allow rapid debug cycle, see results immediately (text outputs, graphics, animations etc…)
  - no need to build and run the code to debug
  - provides timeline feature, useful to inspect how algorithm executes over time
- **Read-Eval-Print-Loop (REPL)**
  - debugging tool for interacting with a running app e.g. playground
  - useful for writing statements that execute immediately
  - can be used directly in Xcode or in Terminal app
- **Interface builder (storyboard editing, view layout and more)**
  - drag-and-drop GUI design of MVCs including transitions between screens
  - use autolayout capabilities to create responsive UI (adaptation to various screen sizes and device orientations)
  - support for many UI design patterns allowing to reduce coding to a minimum
  - live rendering of custom views

Xcode Key Features

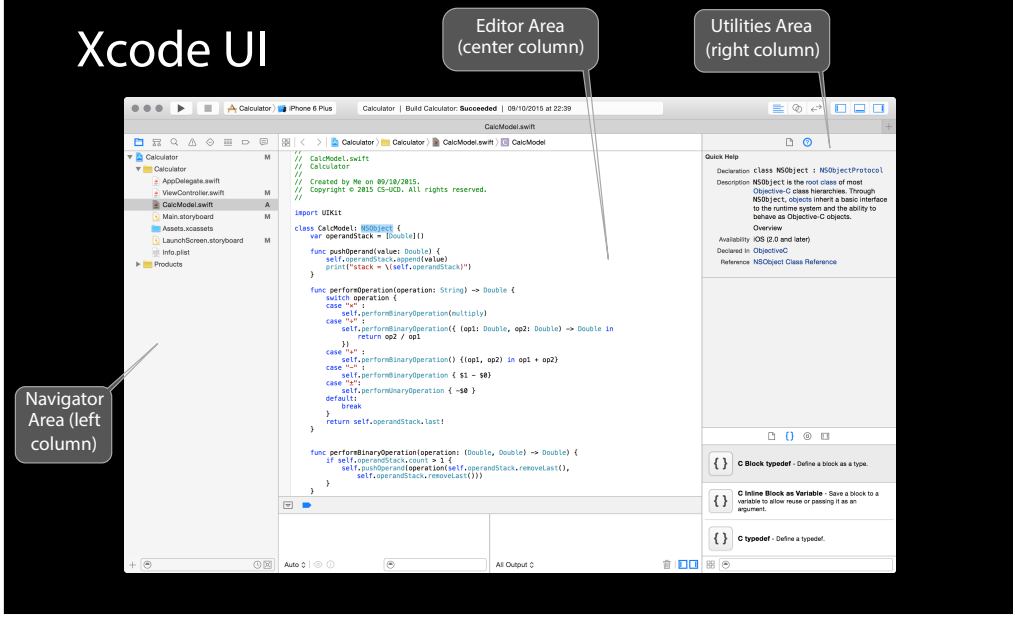
- **iOS simulator**
  - not an emulator
  - test your iOS apps on your Mac
  - provides support for various iPhone and iPad devices including resizable device
- **LLVM compiler**
  - Low Level Virtual Machine, fast open source compile for Swift, Objective–C, C and C++
  - Very well integrated with Xcode
  - LLDB debugger for efficient multicore debugging
- **Assistant editor**
- **Location Simulation**
- **View debugger**
- **Version Control (git support)**
- **Instruments for optimising code**

Xcode UI

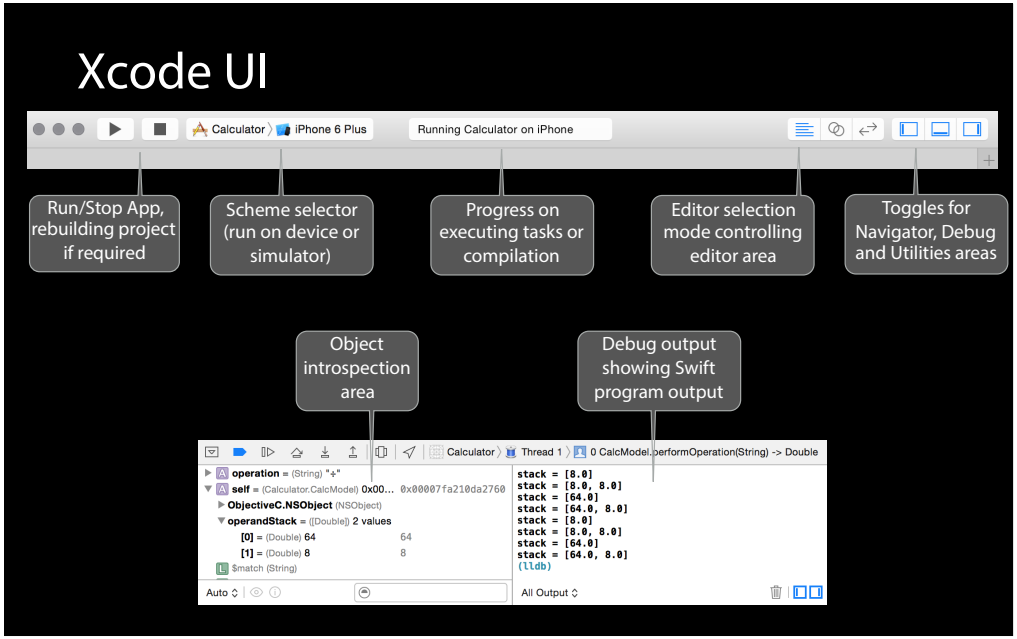
Xcode UI



Xcode UI



Xcode UI



Time for practice