

# SWIFT PROGRAMMING

MPCS 51043  
AUTUMN 2017  
SESSION 3



THE UNIVERSITY OF  
**CHICAGO**

© T.A. BINKOWSKI, 2017

The background features a minimalist abstract design composed of several overlapping curved bands. The primary band is a light grey, with a darker teal band nested within it. A single teal band extends from the left side towards the center. Another teal band is positioned at the top right. The overall effect is organic and fluid.

**NEWS**

# CLASS NEWS

- Unix challenge
- Split the difference on homework regrading

```
cat: *.py: No such file or directory
tabinkowski@Ts-MacBook-Pro ~/Desktop
593 % cat *.swift | grep "//"
// There is always one argument passed,
// that is the file name.
// Read data from this file
    // Data is just a string
    // Split the string into components
//
// Write out a file
//
// Write this text.
// Write the text to the path.
tabinkowski@Ts-MacBook-Pro ~/Desktop
594 % cat *.swift | grep "//" > outfile
```

# CLASS NEWS

```
Desktop — bash — 83x15
cat: *.py: No such file or directory
tabinkowski@Ts-MacBook-Pro ~/Desktop
593 % cat *.swift | grep "//"
// There is always one argument passed, which is the name of the program,
// that is the file name.
// Read data from this file
    // Data is just a string
    // Split the string into components by a newline character "\n"
//
// Write out a file
//
// Write this text.
// Write the text to the path.
tabinkowski@Ts-MacBook-Pro ~/Desktop
594 % cat *.swift | grep //" > outfile.txt
```

- Unix challenge: cat, grep, >

# CLASS NEWS



# Swift

---

ABOUT SWIFT

---

BLOG

---

DOWNLOAD

---

GETTING STARTED

---

DOCUMENTATION

---

MIGRATING TO SWIFT 4

---

SOURCE CODE

---

COMMUNITY

---

CONTRIBUTING

## Dictionary and Set Improvements in Swift 4.0

OCTOBER 4, 2017  [Nate Cook](#)

In the latest release of Swift, dictionaries and sets gain a number of new methods and initializers that make common tasks easier than ever. Operations like grouping, filtering, and transforming values can now be performed in a single step, letting you write more expressive and efficient code.

This post explores these new transformations, using some grocery data for a market as an example. This custom `GroceryItem` struct, made up of a name and a department, will serve as the data type:

- <https://swift.org/blog/dictionary-and-set-improvements/>

# OPTIONALS?!

# OPTIONALS

- Optionals are special type in Swift that enforces safety by being explicit about nothing
- They help prevent you from trying to work with a variable that has no value

```
print("What is your name?")
var name = readLine()
print("Hi, \(name)")
```

```
# What is your name? Andrew
# Hi, Optional(Andrew)
```

# OPTIONALS

- Optionals can represent a value or no value (nil)
- Clarity about what nothing is

```
let x: String?  
let y: Int?  
let z: Double?
```

```
let x: String?  
x = "Hello"  
print(x) // Optional("Hello")
```

# OPTIONALS

- When you are using optionals, you are asking a series of questions:
  - Is the value `nil`?
  - If yes, then do something
  - If no, then unwrap it and copy the value to a new constant

```
...  
if let unwrappedPhone  
    print("The unwrapped phone is \(unwrappedPhone)")  
}  
  
// Unwrap multiple values  
if let unwrappedPhone = phone,  
    unwrappedPad = pad  
    print("The unwrapped phone is \(unwrappedPhone)  
        and the unwrapped pad is \(unwrappedPad)")  
}  
  
// You can also assign  
// (which may be confusing)  
if let phone = phone  
    ...  
}
```

# OPTIONALS

```
//:  
//: # Optional Binding  
//:  
  
if let unwrappedPhone = phone {  
    print("The unwrapped value of phone is: \(unwrappedPhone)")  
}
```

- Optional binding uses `if-let` syntax to unwrap an optional if it is not nil
- Value is assigned to the constant after the `let`

# OPTIONALS

```
// Unwrap multiple values at the same time
if let
    unwrappedPhone = phone,
    unwrappedPad = pad {
    print("The unwrapped value of phone is: \(unwrappedPhone)")
}
```

- Unwrap multiple values
- Will return `nil` if any are `nil`

# OPTIONALS

```
//: ## Forced unwrapping
//: If you already know that a particular optional contains a value, then
//: you can use what is known as forced unwrapping. This means you don't
//: need the if statement to check if the optional contains a value.

var optionalString: String? = "Hello World!"
print("Force unwrapped! \(optionalString!.uppercased())")

//: This is convienent, but eliminates the safety design of Swift
//: ## Implicit unwrapping
var optionalString2: String! = "Hello World!"
print("Implicit unwrapped! \(optionalString2.uppercased())")
```

- Forced unwrap

# OPTIONALS

```
//:  
//: ## Optional Chaining  
//: Optional chaining is a concise way to work with optionals quickly  
//: without using if/let and a conditional block each time.  
  
var maybeString: String? = "The eagle has landed"  
  
// If `maybeString` is not `nil` then `.`uppercaseString` will be evaluated.  
// If it is `nil`, it will return `nil`  
  
if let uppercase = maybeString?.uppercaseString
```

- Optional chaining

# OPTIONALS

```
//:  
//: # Nil-coalescing Operator  
//:  
  
var myOptionalVariable: Int?  
var number: Int = 7  
var number2: Int = myOptionalVariable ?? 0  
  
var result: Int = number + number2  
print(result) // prints: 7
```

- Provide a default value if variable is nil

## OPTIONALS

- `readLine()` returns an optional type
- We need to unwrap it to get the value

```
print("What is your name?")
var input = readLine()
var name = input! ←
print("Hi, \(input)")
```

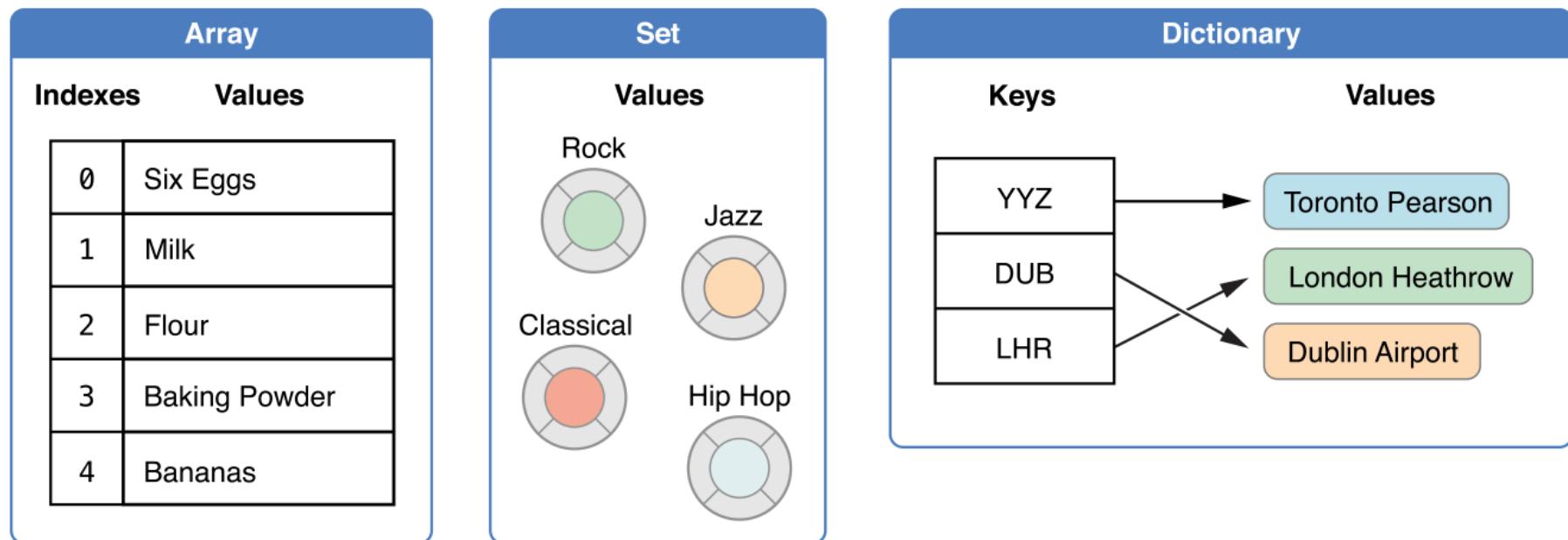
```
func readLine(strippingNewline: Bool = default) -> String? ←
```

# DATA STRUCTURES

# DATA STRUCTURES

- Data structures allow the storage of data in a consistent manner
- Built-in collection types:
  - Arrays, dictionaries, tuples, sets
- Custom types:
  - Define your custom data types specific to your programs specification
  - Structs, classes, enums

# DATA STRUCTURES



# DATA STRUCTURES

- Mutability is determined by the declaration

```
let array = [String]()
var mutableArray = [String]()
```

```
let set = Set<String>()
var mutableSet = Set<String>()
```

# DATA STRUCTURES

- Best practice
  - Immutable collections where collection does not need to change
  - Easier for you to reason about your code
  - Compiler optimization

```
let array = [Int]()
var mutableArray = [Int]()
```

# DATA STRUCTURES

```
// Built-in collection array  
let array = [5, 4, 3, 2, 1]
```

```
// Custom data structure  
struct Person {  
    let name: String  
    let age: Int  
}
```

```
let charlie = Person(name: "Charlie", age: 8)
```

# DATA STRUCTURES

- What is the best type of data structure to use?
- It depends
  - Task
  - Complexity
  - Development time



Sometimes  
simpler is  
better

# ARRAYS

## BASIC COLLECTIONS

- Collections hold groups of data types
- Most basic (and used) data structures
  - In some cases, all you will ever need

```
var shoppingList = ["catfish", "water",
                    "tulips", "blue
                    paint"]
```

```
shoppingList[1] = "bottle of water"
```

# BASIC COLLECTIONS

- Swift standard library has different collections
  - Array
  - Dictionary
  - Set
  - Tuples

```
var shoppingList = ["catfish", "water",  
                    "tulips", "blue  
                    paint"]
```

```
shoppingList[1] = "bottle of water"
```

WE WILL GO OVER THEM ALL  
NEXT WEEK IN DETAIL

# BASIC COLLECTIONS

- Swift has different collections
  - Array
  - Dictionary
  - Set
  - Tuples

```
var arrayOfInts: [Int] = [1, 2, 3]
var arrayOfStrings: [String] =
["A", "B"]
```

```
let arrayOfInts: [Int] = [1, 2, 3]
let arrayOfStrings: [String] =
["A", "B"]
```

```
var emptyArray: [Int] = []
Var emptyArray = [Int]()
```

## BASIC COLLECTIONS

- Access items by index

```
var listOfNumbers = [1, 2, 3, 10, 100]
```

```
listOfNumbers[0] // 1
listOfNumbers[1] // 2
listOfNumbers[2] // 3
listOfNumbers[3] // 10
listOfNumbers[4] // 100
//listOfNumbers[5]// error
```

```
let listSize = listOfNumbers.count
```

## BASIC COLLECTIONS

- Add items with  
`append`

```
var numbers: [Int] = []  
  
numbers.append(1)  
numbers.append(2)  
numbers.append(3)  
numbers.append(4)  
numbers.append(5)  
  
print(numbers)  
// [1, 2, 3, 4, 5]
```

## BASIC COLLECTIONS

- Use `insert` to add a update at a specific index

```
var numbers: [Int] = [1, 2, 3]

numbers.insert(0, at: 0)
// numbers will be [0, 1, 2, 3]

numbers.insert(9, at: 1)
// numbers will be [0, 9, 1, 2, 3]
```

## BASIC COLLECTIONS

- Append using  
+= operator

```
var numbers: [Int] = [1, 2, 3]  
  
numbers += [4, 5, 6]  
//[1, 2, 3, 4, 5, 6]  
  
numbers += [7]  
// [1, 2, 3, 4, 5, 6, 7]
```

# BASIC COLLECTIONS

- Remove items

```
var numbers: [Int] = [1, 2, 3]  
  
numbers.remove(at: 0)  
// numbers will be [2, 3]
```

## BASIC COLLECTIONS

- Change values

```
var numbers: [Int] = [1, 2, 3]

numbers[0] = 7
// numbers will be [7, 2, 3]

numbers[1] = 5
// numbers will be [7, 5, 3]

numbers[2] = 4
// numbers will be [7, 5, 4]
```

## Topics

---

### Creating an Array

In addition to using an array literal, you can also create an array using these initializers.

`init()`

Creates a new, empty array.

`init<S>(S)`

Creates a new instance of a collection containing the elements of a sequence.

`init<S>(S)`

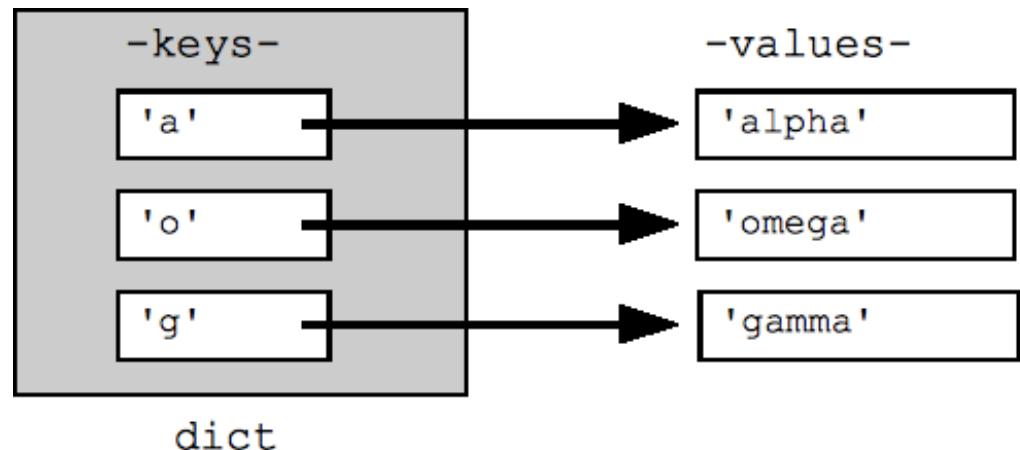
Creates an array containing the elements of a sequence.

- <https://developer.apple.com/documentation/swift/array>

# DICTIONARIES

# DICTIONARIES

- Dictionaries store values that are associated with a key (key-value pair)
  - Perform "lookup" operations
- Dictionaries have different names in different languages
  - Associative Arrays - Perl / PHP
  - Properties or Map or HashMap - Java
  - Property Bag - C# / .Net



# DICTIONARIES

- Dictionaries use `keys` instead of numbers to look up values
- The keys must conform to Hashable protocol 

```
// Dictionary type declaration (rarely used)  
// var hero = Dictionary<String, String>()
```

```
// Shorthand syntax (use this)  
var hero = [String: String]()  
hero["firstName"] = "Bruce"  
hero["lastName"] = "Wayne"  
hero["nickname"] = "Batman"
```

# DICTIONARIES

- The keys must conform to Hashable protocol 

```
// Keys must conform to Hashable
// protocol (eg String, Integer, Bool)
var hero2 = [Int: String]()
hero2[1] = "Bruce"
hero2[2] = "Wayne"
hero2[3] = "Batman"
```

```
// Using a Bool as a key is limited
var hero3 = [Bool: String]()
hero3[true] = "Bruce"
hero3[false] = "Wayne"
```

# DICTIONARIES

- Swift dictionaries are strict

```
var hero = [String: String]()
hero["firstName"] = "Bruce"
hero["lastName"] = "Wayne"
hero["nickname"] = "Batman"
```

```
// These are not allowed because
// they don't conform to the
// defined type
```

```
// hero["age"] = 40
// hero["enemies"] = ["Joker",
//                     "Penguin",
//                     "Poison Ivy"]
```

## DICTIONARIES

```
//: ## Working with Dictionaries ##
var airportCodes: [String: String] = ["YYZ": "Toronto",
                                         "DUB": "Dublin",
                                         "MDW": "Midway"]

print(airportCodes.count)      // 3
print(airportCodes.isEmpty)   // false
```

# DICTIONARIES

Rarely Used

```
// Update the values for a key
airportCodes["MDW"] = "O'Hare"
airportCodes.updateValue("Midway", forKey: "MDW")
```

- Update values for key

# DICTIONARIES

Returns an

```
//: ## Accessing and Modifying Values
print(airportCodes["MDW"])

if let airportName = airports["MDW"] {
    print("The name of the airport is \(airportName).")
} else {
    print("That airport is not in the airports dictionary.")
}
```

"Optional("Midway")\n"

"That airport is not in the airports dictionary.\n"

- Access a value by key return an optional
  - Don't know if there is a value for a key
- Unwrap the value to use it

# DICTIONARIES

- Iteration returns tuple of key and values
- Order is not guaranteed

```
// Iterate through a dictionary
for ac in airportCodes {
    print("\u2022(ac)")
}

for (airportCode, airportName) in airportCodes {
    print("\u2022(airportCode): \u2022(airportName)")
}
```

# DICTIONARIES

Documentation > Swift Standard Li... > Collections > Dictionary    Language: Swift    API Changes: None

## Topics

---

### Creating a Dictionary

In addition to using a dictionary literal, you can also create a dictionary using these initializers.

`init()`

Creates an empty dictionary.

`init(minimumCapacity: Int)`

Creates a dictionary with at least the given number of elements worth of buffer.

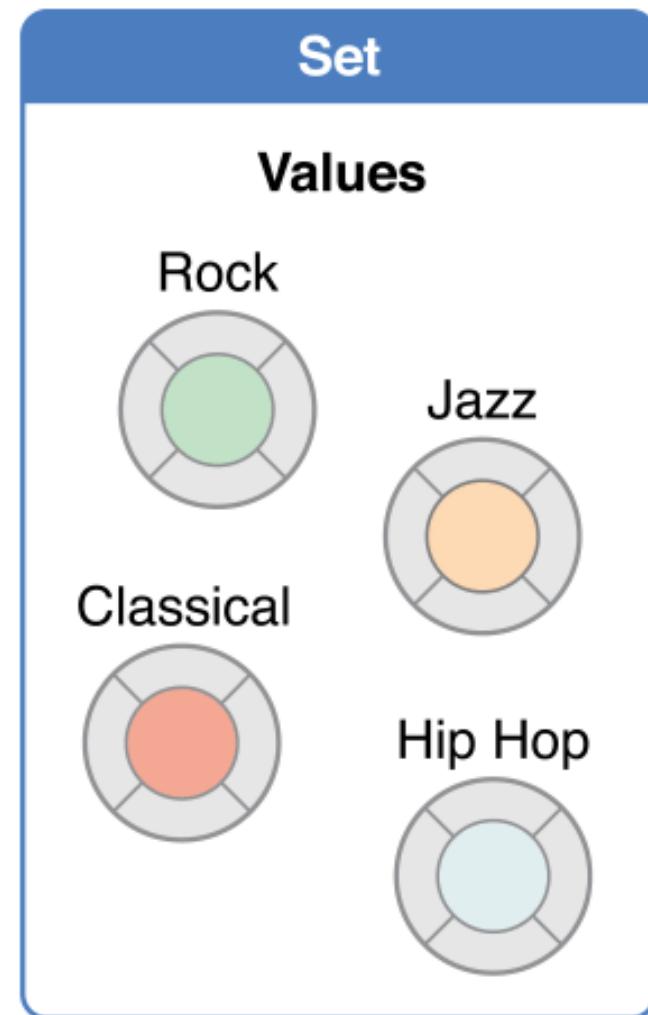
- <https://developer.apple.com/documentation/swift/dictionary>

The background features a minimalist abstract design composed of several overlapping bands. The primary band is a thick, curved teal shape that sweeps from the bottom left towards the top right. Behind it is a thinner, straighter grey band that follows a similar diagonal path. To the right of the teal band is a large, semi-transparent grey area that tapers off towards the top right corner.

**SETS**

# SETS

- Sets store distinct values with no order
- Use a Set instead of an array when order doesn't matter



# SETS

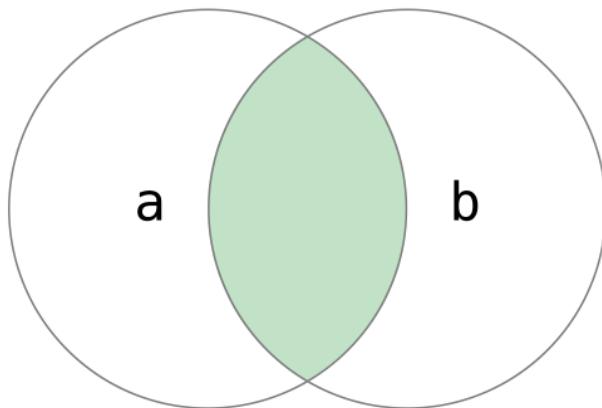
The screenshot shows the Xcode IDE interface. The title bar says "Running session3-collections". The left sidebar shows a project structure with a folder "session3-collections" containing "Dictionaries", "Sources", "Resources", "Optionals", "Sources", "Resources", and a selected "Sets" folder. The main editor area displays the following Swift code:

```
24     print("As far as music goes, I'm not picky.")
25 } else {
26     print("I have particular music preferences.")
27 }
28 // Prints "I have particular music preferences."
29
30 favoriteGenres.insert("Jazz")
31 // favoriteGenres now contains 4 items
32
33 if let removedGenre = favoriteGenres.remove("Rock") {
34     print("\(removedGenre)? I'm over it.")
35 } else {
36     print("I never much cared for that.")
37 }
38 // Prints "Rock? I'm over it."
39
40
41 if favoriteGenres.contains("Funk") {
42     print("I get up on the good foot.")
43 } else {
44     print("Tt's too funky in here.")
```

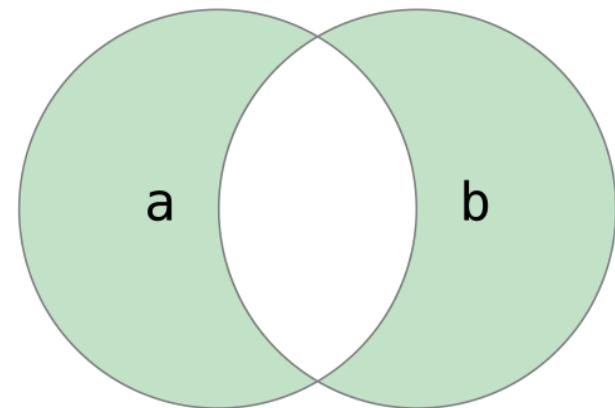
# SETS

- Set have operations to evaluate members of sets

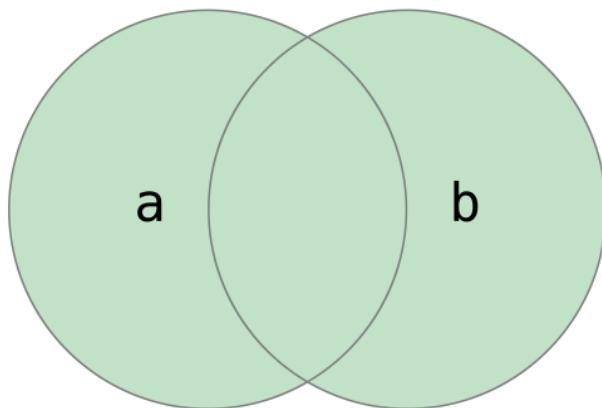
`a.intersection(b)`



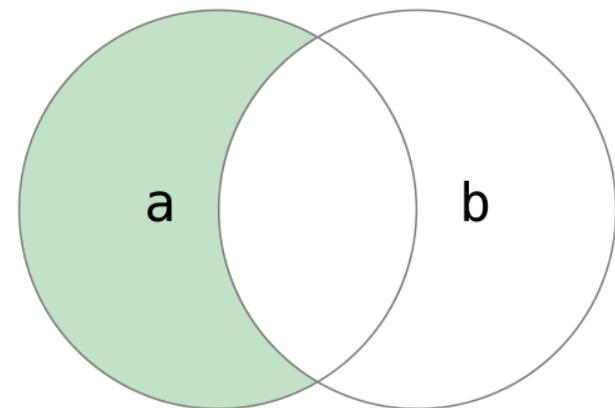
`a.symmetricDifference(b)`



`a.union(b)`



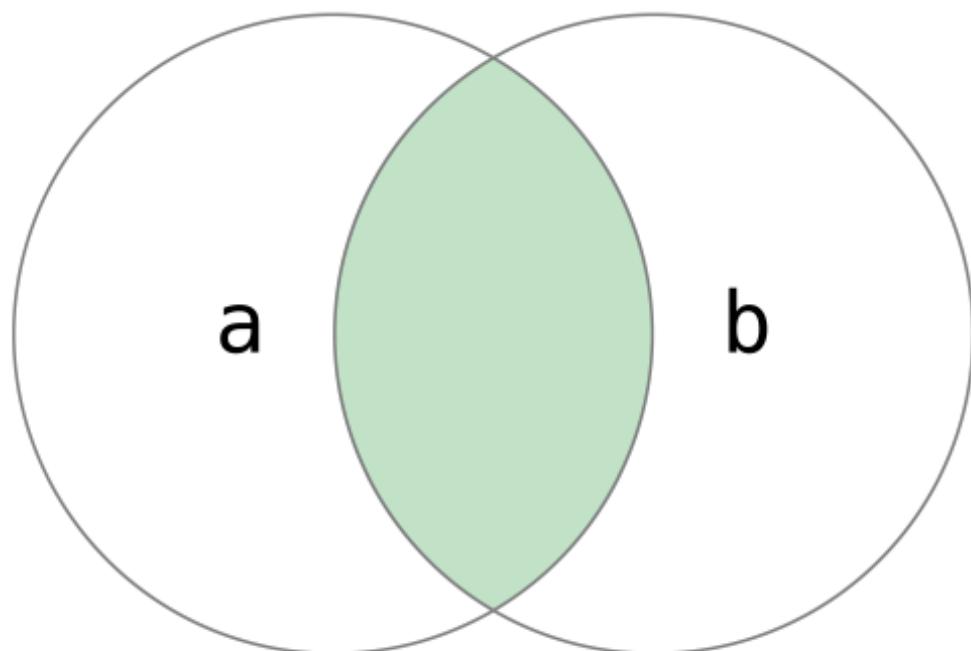
`a.subtracting(b)`



# SETS

- the `intersection(_)` method to create a new set with only the values common to both sets

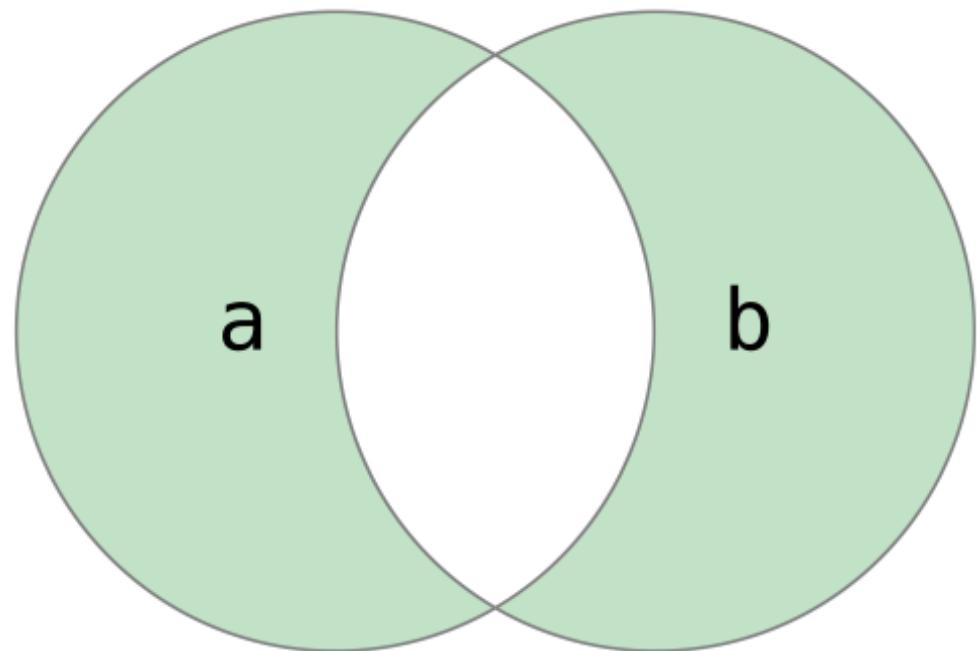
`a.intersection(b)`



# SETS

- the `symmetricDifference(_)` method to create a new set with values in either set, but not both

`a.symmetricDifference(b)`

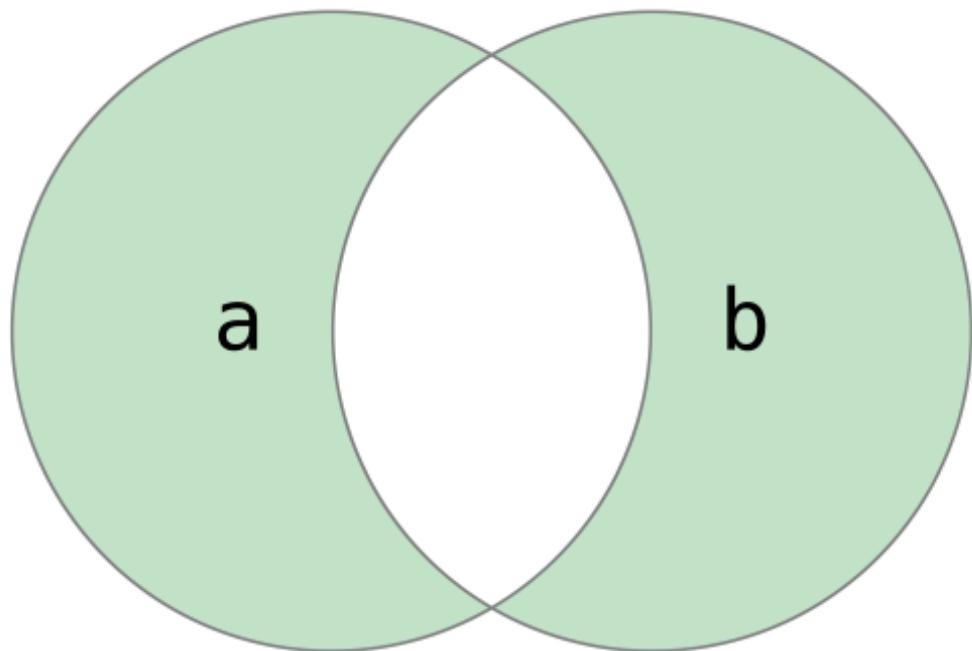


`a.subtracting(b)`

# SETS

- the `symmetricDifference(_)` method to create a new set with values in either set, but not both

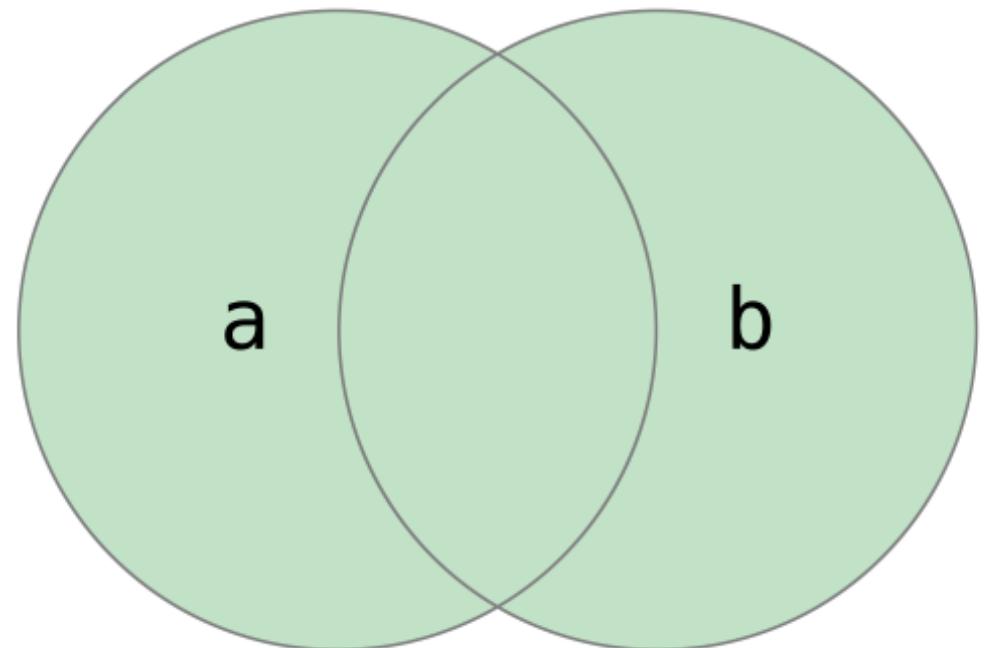
`a.symmetricDifference(b)`



# SETS

- The `union(_)` method to create a new set with all of the values in both sets

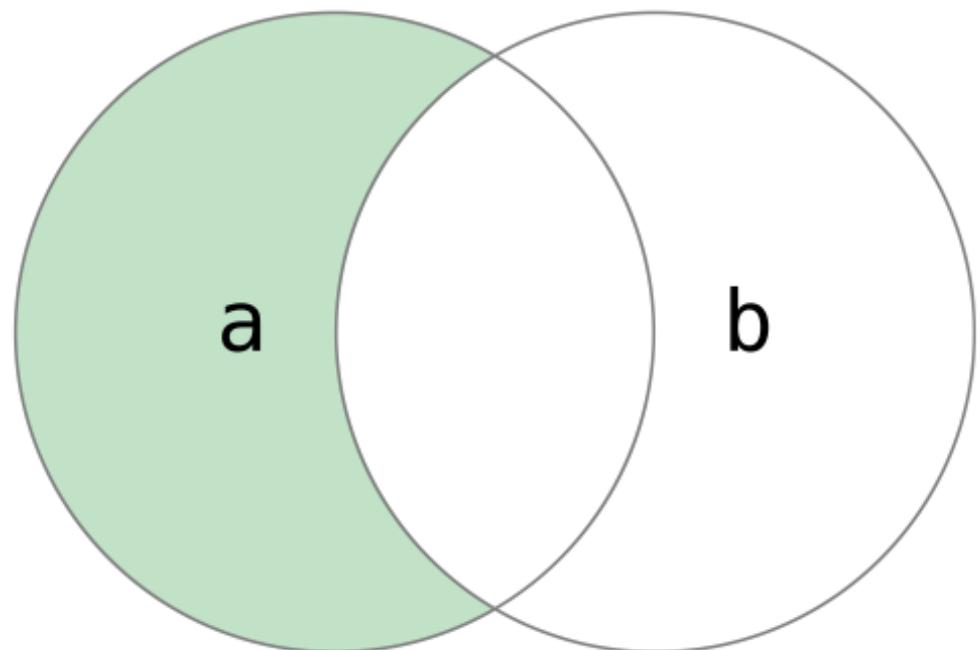
`a.union(b)`



# SETS

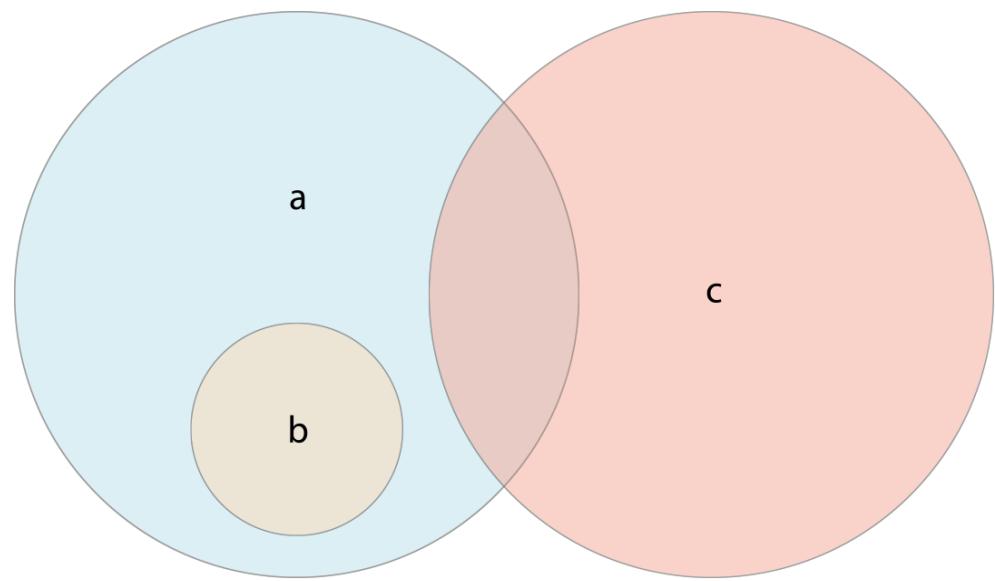
- The subtracting(\_:) method to create a new set with values not in the specified set

a.subtracting(b)



# SETS

- Set `a` is a superset of set `b`
  - `a` contains all elements in b
- `b` is a subset of set a
  - All elements in b are in `a`
- Set `b` and set `c` are disjoint
  - They share no elements in common



## SETS

```
let houseAnimals: Set = ["🐶", "🐱"]  
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]  
let cityAnimals: Set = ["🐦", "🐭"]  
  
houseAnimals == farmAnimals          // false  
houseAnimals.isSubset(of: farmAnimals) // true  
farmAnimals.isSuperset(of: houseAnimals) // true  
farmAnimals.isDisjoint(with: cityAnimals) // true
```

# SETS

`init<S>(S)`

Creates a new set from a finite sequence of items.

---

## Inspecting a Set

`var isEmpty: Bool`

A Boolean value that indicates whether the set has no elements.

`var isEmpty: Bool`

A Boolean value that indicates whether the set is empty.

`var isEmpty: Bool`

A Boolean value indicating whether the collection is empty.

`var count: Int`

- <https://developer.apple.com/documentation/swift/set>

# TUPLES

# TUPLES

```
let myTuple = (1, "b", 3)  
type(of: myTuple)
```

```
(.0 1, .1 "b", .2 3)  
(Int, String, Int).Type
```

- Tuples are another kind of sequence that functions much like a list
- A tuple is a fixed size grouping of elements
  - Elements which are indexed starting at 0

# TUPLES

```
let things = (1, 2, "Blue", "Dog", "🤔", ["a", "b", "c"])
```

```
things.0
things.1
things.2
things.3
things.4
type(of:things.5)
```

```
(.0 1, .1 2, .2 "Blue", .3 "Dog", .4...
```

```
1
2
"Blue"
"Dog"
"🤔"
Array<String>.Type
```

- Tuples can have multiple types

## TUPLES

- Tuples are comparable
- If the first item is equal, go on to next element, and so on, until it finds elements that differ

```
>>> (0, 1, 2) < (5, 1, 2)  
True
```

```
>>> (0, 1, 2000000) < (0, 3, 4)  
True
```

```
>>> ('Jones', 'Sally') < ('Jones',  
'Sam')  
True
```

```
>>> ('Jones', 'Sally') > ('Adams',  
'Sam')  
True
```

# TUPLES

```
var things = (1,2,"Blue","Dog", "🤔", ["a", "b", "c"])

things.0
things.1
things.2
things.3
things.4
type(of:things.5)

// You can change the value as long as the
// type is the same as the initial value
things.0 = 4

things
```

```
(.0 1, .1 2, .2 "Blue", .3 "Dog", .4 "🤔", ["a", "b", "c"])

1
2
"Blue"
"Dog"
"🤔"
Array<String>.Type

4

(.0 4, .1 2, .2 "Blue", .3 "Dog", .4 "🤔", ["a", "b", "c"])
```

- Tuples are mutable (if declared) if the types match at the original index

## TUPLES

- Tuples do not share all the functions as list

```
>>> x = (3, 2, 1)

>>> x.sort()
Traceback:
AttributeError: 'tuple' object has no
attribute 'sort'

>>> x.append(5)
Traceback:
AttributeError: 'tuple' object has no
attribute 'append'

>>> x.reverse()
Traceback:
AttributeError: 'tuple' object has no
attribute 'reverse'
```

# TUPLES

- Why then?
  - Simpler and more efficient in terms of memory use and performance than lists
  - Useful for “temporary variables”
  - Matches data definition
    - (x, y, z) for a coordinate
    - (long, lat) for GPS position

## TUPLES

```
func firstAndLast(numbers: [Int]) -> (Int,Int) {  
    let first = numbers.first!  
    let last = numbers.last!  
    return (first, last)  
}  
  
let numbers = [1,2,3,4,5,6,7,8,9]  
firstAndLast(numbers: numbers)
```

Convenient for  
returning  
multiple values

- Tuples are convenient

# BREAK TIME



# ENUMS

## DEFINING AN ENUMERATION

- An enumeration defines a common type for a group of related values
- Work with those values in a type-safe way within your code

```
enum iOSDeviceType {  
    case iPhone  
    case iPad  
    case appleWatch  
}  
  
let myFavoriteDevice = iOSDeviceType.appleWatch  
print("My favorite iOS device is an \(myFavoriteDevice)")
```

## DEFINING AN ENUMERATION

- Define a new enumeration using the `enum` keyword followed by it's name
- The member values are introduced using the `case` keyword

Name

```
enum iOSDeviceType {  
    case iPhone  
    case iPad  
    case appleWatch  
}
```

Members

```
let myFavoriteDevice = iOSDeviceType.appleWatch  
print("My favorite iOS device is an \(myFavoriteDevice)")
```

# DEFINING AN ENUMERATION

- Best practice to use capital first letter for enum and lowercase for the members
- Enum is a custom type; all types start with capital in Swift standard lib

```
enum CompassPoint {  
    case north  
    case south  
    case east  
    case west  
}
```

Name

Members/  
Enumeration  
cases

# DEFINING AN ENUMERATION

```
enum Planet {  
    case mercury, venus, earth, mars, jupiter, saturn, uranus, neptune  
}
```

- For the lazy, declare multiple cases on a single line

## DEFINING AN ENUMERATION

- If the type of an enumeration is known or can be inferred then you can use the dot syntax for members
- Some UIKit Enums are really, really, really, really long

```
import Foundation

enum iOSDeviceType {
    case iPhone
    case iPad
    case appleWatch
}

let myFavoriteDevice = iOSDeviceType.appleWatch

// Inferred enum can use dot syntax
let myLeastFavorableDevice: iOSDeviceType = .iPad
```

It already knows its an  
iOSDeviceType from  
Declaration

## DEFINING AN ENUMERATION

- Use switch statement to match enumerated values

```
//:  
//: # Using Enumerations  
//:  
  
var directionToHead: CompassPoint = .south  
  
switch directionToHead {  
case .north:  
    print("Lots of planets have a north")  
case .south:  
    print("Watch out for penguins")  
case .east:  
    print("Where the sun rises")  
case .west:  
    print("Where the skies are blue")  
}
```

## DEFINING AN ENUMERATION

- Switch statements must be exhaustive
- Error here for no `east` case

```
enum CompassPoint {  
    case north  
    case south  
    case east  
    case west  
}
```

```
// Let's go on a trip  
var directionToHead: CompassPoint = .south  
  
switch directionToHead {  
case .north:  
    print("Lots of planets have a north")  
case .south:  
    print("Watch out for penguins")  
case .west:  
    print("Where the skies are blue")  
}  
?
```

## DEFINING AN ENUMERATION

- Switch statements must be exhaustive
- Error here for no `east` case

```
enum CompassPoint {  
    case north  
    case south  
    case east  
    case west  
}
```

```
// Let's go on a trip  
var directionToHead: CompassPoint = .south  
  
switch directionToHead {  
case .north:  
    print("Lots of planets have a north")  
case .south:  
    print("Watch out for penguins")  
case .west:  
    print("Where the skies are blue")  
}  
?
```

## DEFINING AN ENUMERATION

- Use `default` case to match everything

```
// Let's go on a trip
var directionToHead: CompassPoint = .south

switch directionToHead {
case .north:
    print("Lots of planets have a north")
default:
    print("Use Google maps")
}
```

# ASSOCIATED VALUES

- Enums can store associated values
  - The value type can be different for each member
  - Can be String, Int, Float, Char

Associated

```
enum WebService: String {  
    case Post = "http://something/upload"  
    case Request = "http://something/request"  
}  
  
var postUrl: WebService = .Post  
  
// Use the rawValue property to access the raw  
// value of an enumeration member  
print("Post URL: \(postUrl.rawValue)")  
  
// If the type is inferrable, you can access the  
// value using dot syntax.  
postUrl = .Request  
print("Request URL: \(postUrl.rawValue)")
```

# ASSOCIATED VALUES

- `raw value` will be the same type for members
- Values for each member must be unique

```
enum WebService: String {  
    case Post = "http://something/upload"  
    case Request = "http://something/request"  
}  
  
var postUrl: WebService = .Post  
  
// Use the rawValue property to access the raw  
// value of an enumeration member  
print("Post URL: \(postUrl.rawValue)")  
  
// If the type is inferrable, you can access the  
// value using dot syntax.  
postUrl = .Request  
print("Request URL: \(postUrl.rawValue)")
```

Raw value

Access the associated value with `.rawValue`

## ASSOCIATED VALUES

- When integers are used they auto increment when not defined for a member

```
enum CompassPoint: Int {  
    case north = 1  
    case south  
    case east  
    case west  
}
```

Doesn't have to

Autoincrement

## ASSOCIATED VALUES

- `rawValues` can be anything in any order as long as they are the same type

```
enum CompassPoint: Int {  
    case north = 1  
    case south = 100  
    case east = 3  
    case west = 4  
}
```

## ASSOCIATED VALUES

```
enum WebService: String {
    case Post = "http://something/upload"
    case Request = "http://something/request"
}

// Set an enum value using rawValue
let requestURL = WebService(rawValue: "http://something/request")
print(requestURL!)
```



Webservice.Request

- Set an Enum using `rawValue`

## ASSOCIATED VALUES

```
enum SchoolSupplies {  
    case book(String)  
    case money(Double)  
    case lunchOrder(Int)  
}
```

```
let supplies: SchoolSupplies = .money(100)  
let book: SchoolSupplies = .book("War and Peace")  
let lunch: SchoolSupplies = .lunchOrder(5)
```

Discriminated  
Union...of course



- The 20% you'll never use; better ways of doing this with other data structures

# ERROR TYPE

# ERROR TYPE

- In Swift, errors are represented by values of types that conform to the `ErrorType` protocol
- This empty protocol indicates that a type can be used for error handling

```
errorType :>  
    ring)
```

## ERROR TYPE

```
///  
func validateUser(username: String, password: String)  
//  
guard password.characters.count > 0 else {  
    throw UserValidationError.Empty  
}  
  
//  
guard password.characters.count >= 6 else {  
    throw UserValidationError.MinimumLength  
}  
  
//  
guard password == "password" else {  
    throw UserValidationError.Incorrect(message: "Wrong password")  
}  
  
// Lookup userId based on name and password
```

```
//  
// Custom ErrorType  
//  
enum UserValidationError: ErrorType {  
    case Empty  
    case MinimumLength  
    case Incorrect(message: String)  
}
```

# ERROR TYPE

```
try self.validateUser("user@email.com", password: "123456")
showAlert("Successful")

    in UserValidationError.Empty {
        showAlert("Form Empty")
        return

    in UserValidationError.MinimumLength {
        showAlert("Password too short")
        return

    in UserValidationError.Incorrect(let message) {
        showAlert(message)
        return

    }
}
Swift wants error handling to account an undefined error
```

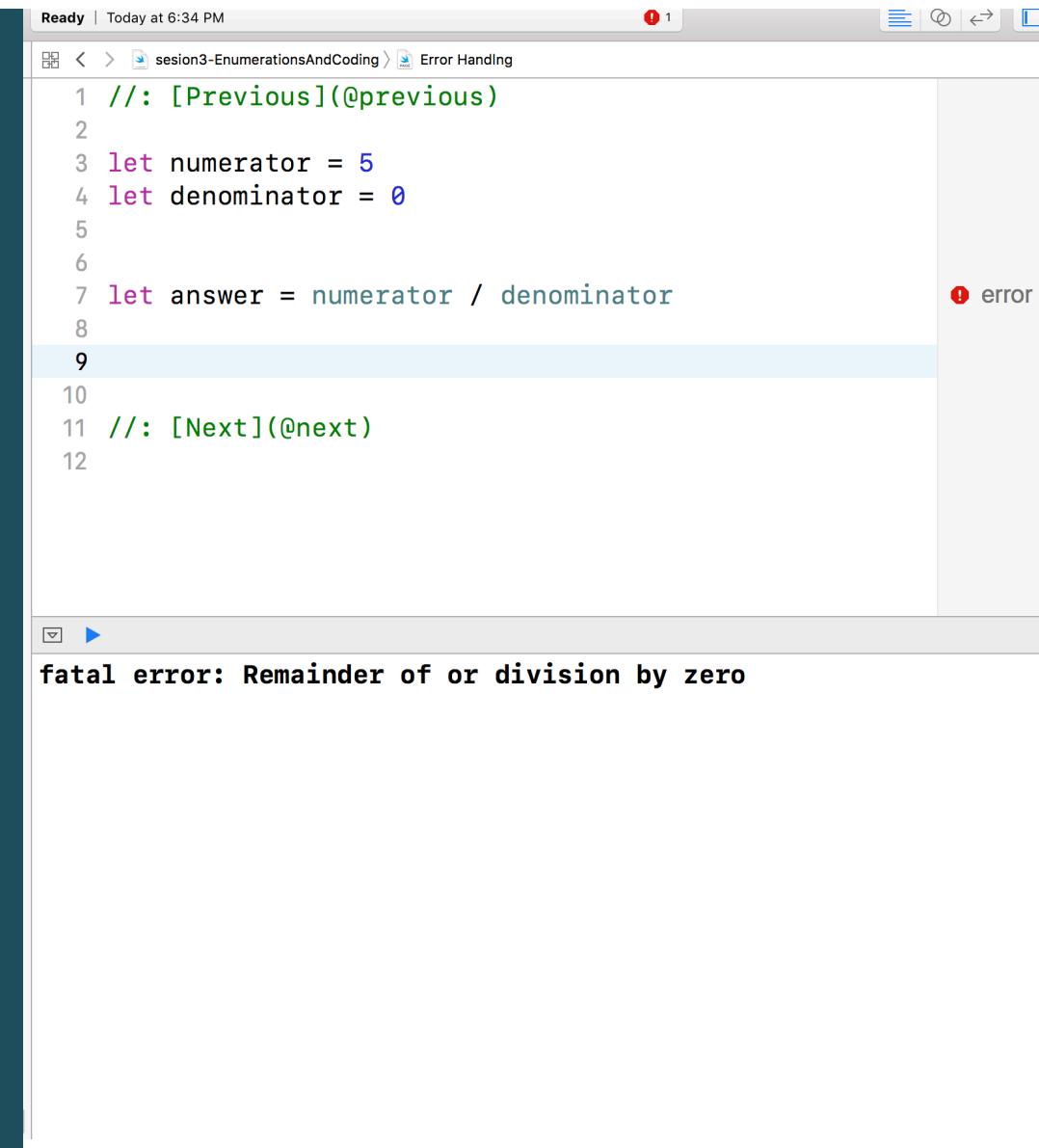
```
// // Custom ErrorType
// enum UserValidationError: ErrorType {
//     case Empty
//     case MinimumLength
//     case Incorrect(message: String)
// }
```



# ERROR HANDLING

# ERROR HANDLING

- Error handling is the process of responding to and recovering from error conditions in your program
- Allow your program to run when it would normally crashed



The screenshot shows a code editor window titled "Ready | Today at 6:34 PM". The file is named "Error Handling" and is located in a folder named "sesion3-EnumerationsAndCoding". The code itself is:

```
1 //: [Previous](@previous)
2
3 let numerator = 5
4 let denominator = 0
5
6
7 let answer = numerator / denominator
8
9
10
11 //: [Next](@next)
12
```

A red exclamation mark icon in the top right corner indicates an error. Below the code, a message box displays the error: "fatal error: Remainder of or division by zero".

# ERROR HANDLING

The screenshot shows a Xcode interface with the following details:

- Toolbar:** Ready | Today at 6:35 PM
- File Navigator:** sesion3-EnumerationsAndCoding (selected), showing files like Enumerations, Error Handling, etc.
- Code Editor:** A Swift file named "Error Handling". The code is:

```
1 //: [Previous](@previous)
2
3 let numerator = 5
4 let denominator = 0
5
6 if denominator != 0 {
7     let answer = numerator / denominator
8     print(answer)
9 }
10
11
12 //: [Next](@next)
```
- Annotations:**
  - A yellow callout bubble points to the condition `if denominator != 0 {` with the text "No Error by testing condition".
  - A yellow callout bubble points to the line `print(answer)` with the text "We don't really have any feedback on what happened; unexpected operation from this point on".
- Right Margin:** Shows line numbers 5 and 0, and icons for copy/paste and zoom.

# ERROR HANDLING

The screenshot shows an Xcode playground window titled "Error Handling". The code attempts to divide 5 by 0:`//: [Previous](@previous)  
let numerator = 5  
let denominator = 0  
var answer: Double  
if denominator != 0 {  
 answer = Double(numerator / denominator)  
}  
print("The answer is \(answer)")`A yellow callout bubble points to the division operation `numerator / denominator` with the text "No Error by testing condition". Another yellow callout bubble points to the "Playground execution failed" message at the bottom with the text "We don't really have any feedback on what happened; unexpected operation from this point on".

# ERROR HANDLING

The screenshot shows a Xcode interface with a project named "sesion3-EnumerationsAndCoding". The current file is "Error Handling". The code attempts to divide 5 by 0 and prints the result:

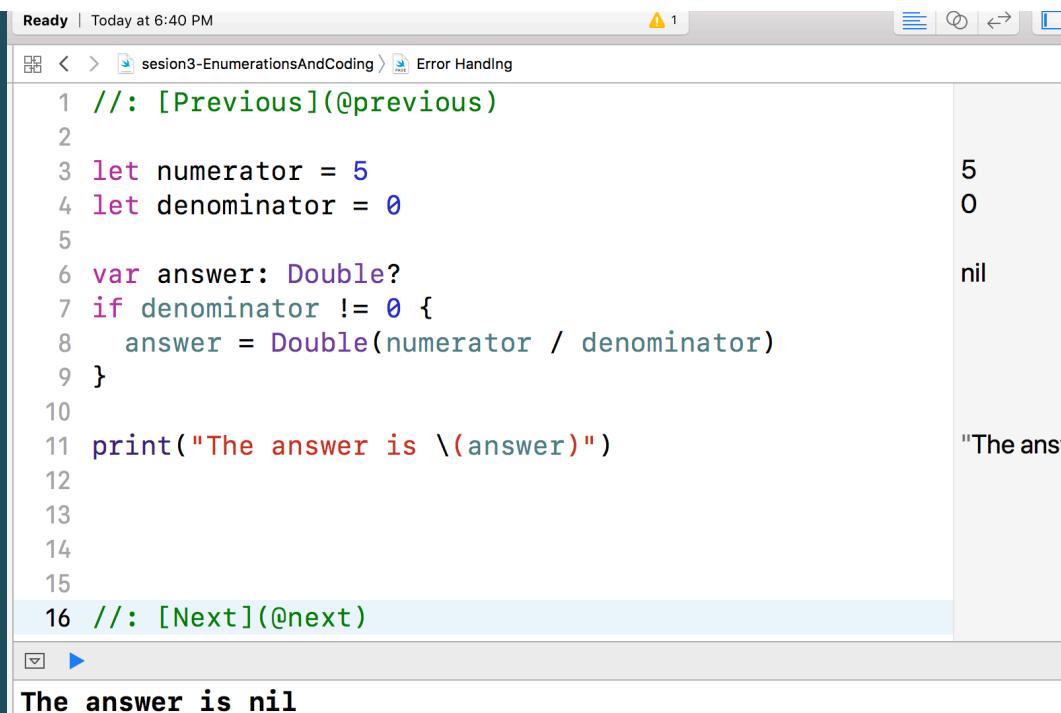
```
//: [Previous](@previous)
let numerator = 5
let denominator = 0
var answer: Double?
if denominator != 0 {
    answer = Double(numerator / denominator)
}
print("The answer is \(answer)")

//: [Next](@next)
```

The output of the print statement is "The answer is nil". A yellow callout box on the left says: "We don't really have any feedback on what happened; just that its nil". An arrow from this box points to the word "nil" in the output. Another yellow callout box on the right says: "We can make this an optional". An arrow from this box points to the line "var answer: Double?".

# ERROR HANDLING

- Swift provides first-class language support for throwing, catching, propagating, and manipulating recoverable errors at runtime
- Provides a systematic and informative way to handle errors



The screenshot shows a Xcode editor window with the following details:

- Top Bar:** Ready | Today at 6:40 PM, 1 warning.
- Project Navigator:** Shows a single file: session3-EnumerationsAndCoding > Error Handling.
- Code Editor:** Displays the following Swift code:

```
1 //: [Previous](@previous)
2
3 let numerator = 5
4 let denominator = 0
5
6 var answer: Double?
7 if denominator != 0 {
8     answer = Double(numerator / denominator)
9 }
10
11 print("The answer is \(answer)")
12
13
14
15
16 //: [Next](@next)
```
- Output Area:** Shows the result of the `print` statement: "The answer is nil".
- Right Margin:** Shows line numbers 5, 0, and nil, along with a partial string "The ans".

# ERROR TYPES

## ERRORTYPE ENUM

- Swift errors are represented by special enums values types of `Error`
- Conform to `Error` protocol

Error type

```
enum DivisionError: Error {  
    case divideByZero  
    case nonNumeric  
}
```

# ERRORTYPE ENUM

- iOS APIs are full of custom error enums
- You can write your own

outError

## AVAudioPlayer

Inherits from: [NSObject](#)

Conforms to: [NSObject](#)

Framework: [AVFoundation](#) in iOS 2.2 and later. [More related items...](#)

### - [initWithContentsOfURL:error:](#)

Initializes and returns an audio player for playing a designated sound file.

#### Declaration

[SWIFT](#)

```
init(contentsOfURL url: NSURL) throws
```

[OBJECTIVE-C](#)

```
- (instancetype)initWithContentsOfURL:(NSURL *)url  
error:(NSError * _Nullable *)out
```

#### Parameters

<code>url</code>	A URL identifying the sound file to play. The audio data must be in one of the supported formats. For a list of supported formats, see <a href="#">Using Audio</a> in <a href="#">Multimedia Programming Guide</a> .
------------------	--

<code>outError</code>	If an error occurs, upon return the <code>NSError</code> object describes the error.
-----------------------	--

#### Return Value

On success, an initialized `AVAudioPlayer` object. If `nil`, the `outError` parameter describes the problem.

## ERRORTYPE ENUM

- Throwing an error lets you indicate that something unexpected happened and the normal flow of execution can't continue

```
...  
enum DivisionError: Error {  
  case divideByZero  
  case conversion  
}  
  
throw DivisionError.divideByZero
```

throw an error

# ERROR TYPE ENUM

- `throw` keyword passes an error to the statement that called the function
- `try` keyword used with throws

```
func canThrowErrors() throws -> String  
try canThrowError()
```

```
func cannotThrowErrors() -> String
```

# ERRORTYPE ENUM

```
enum DivisionError: Error {
    case divideByZero
}

func safeDivision(numerator: Int, denominator: Int) throws -> Double? {
    // Check all conditions that would result in an error
    if denominator != 0 {
        throw DivisionError.divideByZero
    }
    return Double(numerator / denominator)
}

do {
    _ = try safeDivision(numerator:10, denominator:0)
} catch DivisionError.divideByZero {
    print("Error: Division by 0")
}
```



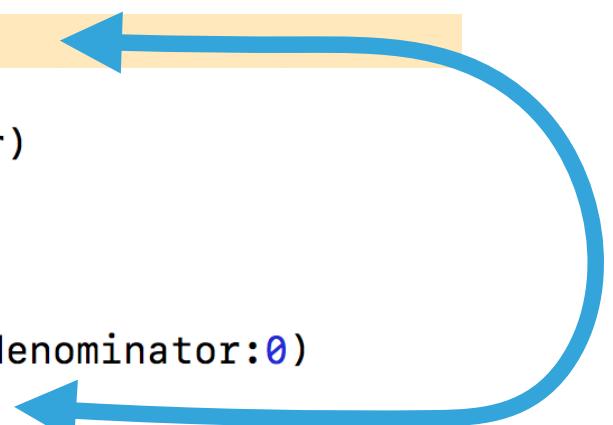
# ERRORTYPE ENUM

```
enum DivisionError: Error {
    case divideByZero
}

func safeDivision(numerator: Int, denominator: Int) throws -> Double? {

    // Check all conditions that would result in an error
    if denominator != 0 {
        throw DivisionError.divideByZero
    }
    return Double(numerator / denominator)
}

do {
    _ = try safeDivision(numerator:10, denominator:0)
} catch DivisionError.divideByZero {
    print("Error: Division by 0")
}
```



# ERRORTYPE ENUM

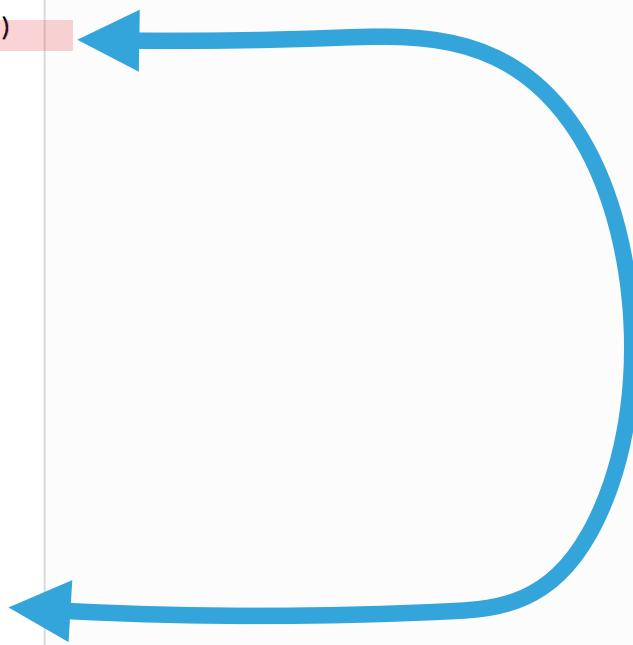
```
do {
    let userId = try self.validateUser("user@email.com", password: "123456")
    print("Login was successfull: \(userId)")
}
catch UserValidationError.Empty {
    //
}
catch UserValidationError.MinimumLength {
    //
}
catch UserValidationError.Incorrect {
    //
}
catch {
    // Swift wants error handling to be exhaustive, so you must take into
    // account an undefined error
}

func validateUser(username: String, password: String) throws -> String {
    // throw UserValidationError.Empty

    // throw UserValidationError.MinimumLength

    // throw UserValidationError.Incorrect

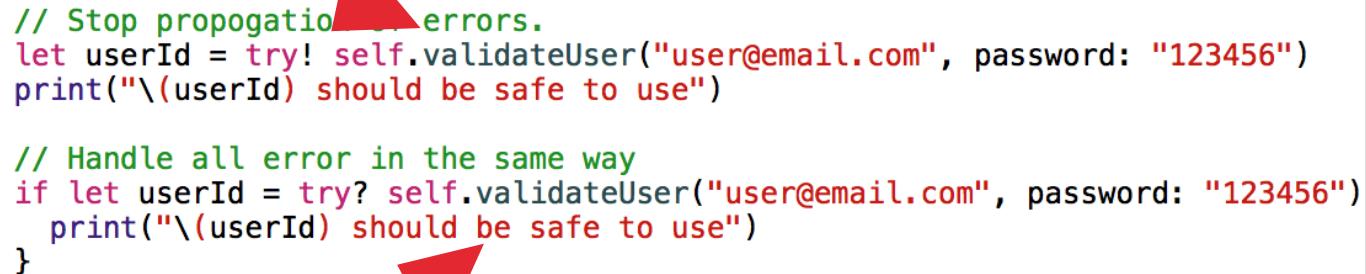
    // If you made it here, you can return:
    // let userId = //do something here
}
```



# ERRORTYPE ENUM

- Alternative ways of handling errors
  - ! Ignore
  - ? Convert to optional

```
// Stop propagation of errors.  
let userId = try! self.validateUser("user@email.com", password: "123456")  
print("\(userId) should be safe to use")  
  
// Handle all error in the same way  
if let userId = try? self.validateUser("user@email.com", password: "123456")  
    print("\(userId) should be safe to use")  
}
```

Two red arrows point from the left towards the 'try!' and 'try?' keywords in the code, highlighting them.

## ERRORTYPE ENUM

- If an error is thrown while evaluating the try? expression, the value of the expression is `nil`

```
func someThrowingFunction() throws -> Int {  
    // ...  
}  
  
let x = try? someThrowingFunction()  
  
let y: Int?  
do {  
    y = try someThrowingFunction()  
} catch {  
    y = nil  
}
```

## ERRORTYPE ENUM

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

- Use `try!` to disable throwing an error
- This does not mean that the results is safe

# ERRORTYPE ENUM

► Resources

```
enum UserValidationError: Error {
    case empty
    case minLength
    case incorrect(message: String)
}
```

## Login Handling

```
/// Conduct basic validation on the username and password
/// - Returns: A `String` of the userId
/// - Throws: A `UserValidationError`
///
func validateUser(_ username: String, password: String) throws -> String {

    guard password.characters.count > 0 else {
        throw UserValidationError.empty
    }

    guard password.characters.count >= 6 else {
        throw UserValidationError.minimumLength
    }
}
```



# FILES AND PARSING

# FILES AND PARSING

- Applications need to persist data between sessions
- Reading and writing data to disk

```
<Books>
  <Book ISBN="0553212419">
    <title>Sherlock Holmes: Complete Novels...
    <author>Sir Arthur Conan Doyle</author>
  </Book>
  <Book ISBN="0743273567">
    <title>The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
  </Book>
  <Book ISBN="0684826976">
    <title>Undaunted Courage</title>
    <author>Stephen E. Ambrose</author>
  </Book>
  <Book ISBN="0743203178">
    <title>Nothing Like It In the World</title>
    <author>Stephen E. Ambrose</author>
  </Book>
</Books>
```

# FILES AND PARSING

- Files contain different types of data
  - Structured format (csv, XML, JSON, HTML)
  - Text (.txt)
  - Binary (.docx, .sql)
- An application can use any of these file types to save data

```
<Books>
  <Book ISBN="0553212419">
    <title>Sherlock Holmes: Complete Novels...
    <author>Sir Arthur Conan Doyle</author>
  </Book>
  <Book ISBN="0743273567">
    <title>The Great Gatsby</title>
    <author>F. Scott Fitzgerald</author>
  </Book>
  <Book ISBN="0684826976">
    <title>Undaunted Courage</title>
    <author>Stephen E. Ambrose</author>
  </Book>
  <Book ISBN="0743203178">
    <title>Nothing Like It In the World</title>
    <author>Stephen E. Ambrose</author>
  </Book>
</Books>
```

# FILES AND PARSING

- Different formats have different properties
  - Size
  - Privacy
  - Human readable

## JSON

```
{  
  "siblings": [  
    {"firstName": "Anna", "lastName": "Clayton"},  
    {"lastName": "Alex", "lastName": "Clayton"}  
  ]  
}
```

## XML

```
<siblings>  
  < sibling >  
    < firstName >Anna</firstName>  
    < lastName >Clayton</lastName>  
  </ sibling >  
  < sibling >  
    < firstName >Alex</firstName>  
    < lastName >Clayton</lastName>  
  </ sibling >  
</ siblings >
```

# FILES AND PARSING

- Swift has built-in functions to help read text-based files
- Reading some file types may require special libraries

```
import Foundation

// Read data from this file
let path = "./data.txt"

do {
    let data = try String(contentsOfFile: path, encoding: String.Encoding.utf8)
    // Data is just a string
    print(data)

    // Split the string into components by a newline character "\n"
    for line in data.components(separatedBy: CharacterSet.newlines) {
        print("> Line: \(line)")
    }
} catch {
    print("Error: \(error)")
}
```

# FILES AND PARSING

- One approach of many, but the most straightforward

```
// Read data from this file
let path = "./data.txt"

do {
    let data = try String(contentsOfFile: path,
                           encoding: String.Encoding.utf8)
    // Data is just a string
    print(data)

    // Split the string into components by a newline character "\n"
    for line in data.components(separatedBy: CharacterSet.newlines) {
        print("> Line: \(line)")
    }
} catch {
    print("Error:\(error)")
}
```

# FILES AND PARSING

- Files have special (invisible) characters used in formatting
  - `'\n` - "newline" to indicate when a line ends
  - `'\t` - tab
  - Use them to determine line breaks

1	a ↴
2	b ↴
3	c ↴
4	d ↴
5	e ↴
6	f ↴
7	g ↴
8	h ↴
9	i ↴
10	j ↴
11	k ↴

# FILES AND PARSING

- Pay attention to newline characters when printing
- You are reading them in from the file

End of file  
line

---

1	a ↴
2	b ↴
3	c ↴
4	d ↴
5	e ↴
6	f ↴
7	g ↴
8	h ↴
9	i ↴
10	j ↴
11	k ↴
12	

# FILES AND PARSING

- We can read the whole file into a single string
  - \n are read in as well

```
let data = try String(contentsOfFile: path,  
.....encoding: String.Encoding.utf8)  
// Data is just a string  
print(data)  
  
// Split the string into components by a newline character "\n"  
for line in data.components(separatedBy: CharacterSet.newlines) {  
print("> Line: \(line)")  
}
```

# FILES AND PARSING

- Clean up your data using string functions
- 'Whitespace' means characters you can not see
  - \t,\n,\ ,etc.

Documentation > Swift Standard Li... > Strings and Text > String Language: Swift API Changes: Show ▾

## Inserting Characters

func `insert`(Character, `at`: String.Index)

Inserts a new character at the specified position.

func `insert`(Character, `at`: String.Index)

Inserts a new element into the collection at the specified position.

func `insert`<S>(contentsOf: S, `at`: String.Index)

Inserts a collection of characters at the specified position.

func `insert`<C>(contentsOf: C, `at`: String.Index)

Inserts the elements of a sequence into the collection at the specified position.

func `reserveCapacity`(String.CharacterView.IndexDistance)

Prepares the collection to store the specified number of elements, when doing so is appropriate for the underlying type.

## Replacing Substrings

func `replaceSubrange`<C, R>(R, `with`: C)

Replaces the specified subrange of elements with the given collection.

func `replaceSubrange`<C>(Range<String.Index>, `with`: C)

Replaces the text within the specified bounds with the given characters.

<https://developer.apple.com/documentation/swift/string>

Removes and returns the character at the specified position

# FILES AND PARSING

- Cleaning up data is a fundamental problem in computer science
- There are many tools and workflows to accomplish a task
  - Sometimes it is easier to clean up your data before inputting it into a program
  - Sometimes you will write two programs (one to clean, one to process)

# ASSIGNMENT

# ASSIGNMENT

MPCS  
51043

## Assignment

---

Assignment 1 is due October 7, 2017 at 8:59am.

- Sign up for a GitHub account (if you don't have one).
- Sign up for Slack and [join](#) our workspace.
- Post a introductory comment to the [#general](#) channel on Slack. Tell everyone a little about yourself and your background (e.g. full/part time student, undergraduate degree, etc.). Please upload a picture of yourself so we know you who are. Hilarious pictures are encouraged.
- Setup your development environment by downloading Xcode and Atom. Use the latest version on Xcode that is on the Mac App Store.
- Review the [Swift Guided Tour Playground](#) Apple. Stop reading after [Functions and Closures](#) chapter.  
<http://uchicago.codes>
- There are many resources for learning Swift listed above. Please

# THE END

MPCS 51043  
AUTUMN 2017  
SESSION 3



THE UNIVERSITY OF  
**CHICAGO**

© T.A. BINKOWSKI, 2017

# UNIT TESTING

# GENERICS

# LISTS

# LISTS

- List constants are surrounded by square brackets
- The elements in the list are separated by commas

```
# List of strings  
friends = [ 'Nick', 'Jane', 'Rachel']  
  
# List of integers  
favorite_numbers = [ 1, 2, 3, 4, 5]  
  
# List iteration  
dogs = [ 'spot', 'underdog', 'snooopy']  
for dog in dogs:  
    print dog,  
  
>>> spot, underdog, snooopy
```

# LISTS

- A list element can be any Python object
- Even another list

```
# List of strings
cats = [ 'garfield', 'lola', 'scaredy']
dogs = [ 'spot', 'underdog', 'snooopy']

# List of lists
pets = [ cats, dogs ]

>>> pets
[['garfield', 'lola', 'scaredy'],
 ['spot', 'underdog', 'snooopy']]
```

## LISTS

- Iterating a list of lists

```
cats = [ 'garfield', 'lola', 'scaredy']
dogs = [ 'spot', 'underdog', 'snooopy']
pets = [ cats, dogs ]
```

```
# Loop through the list of lists
for pet in pets:
```

```
# The iteration variable is a list
for animal in pet:
    print animal,
```

```
>>> [[ 'garfield', 'lola', 'scaredy'],
      [ 'spot', 'underdog', 'snooopy']]
```

# LISTS

```
for pet in pets:  
    print type(pet)  
  
if isinstance(pet, list):  
    for animal in pet:  
        print animal,
```

Remember a list can contain  
different types

```
<type 'list'>  
['garfield', 'lola', 'scaredy']  
<type 'list'>  
['spot', 'underdog', 'snooopy']
```

# LISTS

- Access the elements of list by their index

```
cats = [ 'garfield', 'lola', 'scaredy']
dogs = [ 'spot', 'underdog', 'snooopy']
pets = [cats, dogs]

print pets[0]          # cats list
print pets[0][0]        # garfield
print pets[0][1]        # lola

print pets[1]          # dogs list
print pets[1][1]        # underdog
```

# LISTS

- If an index has a negative value, it counts backward from the end

```
cats = [ 'garfield', 'lola', 'scaredy']

print cats[-1] # scaredy

print cats[-2] # lola

print cats[-3] # garfield
```

# LISTS

- Any integer expression can be used as an index
- IndexError if you try to read or write an element that does not exist

```
cats = [ 'garfield', 'lola', 'scaredy']  
index = 4
```

```
print cats[index]
```

```
Traceback (most recent call last):  
  File "workspace.py", line 4, in  
    <module>  
      cats[4]  
IndexError: list index out of range
```

# LISTS

- The `range()` function returns a list of numbers that range from zero to one less than the parameter
- Take 1 or 2 parameters

```
for i in range(len(cats)):  
    print "#", i,"->",cats[i]  
# 0 -> garfield  
# 1 -> lola  
# 2 -> scaredy  
  
for i in range(0,len(cats)):  
    print "#", i,"->",cats[i]  
# 0 -> garfield  
# 1 -> lola  
# 2 -> scaredy  
  
for i in range(1,len(cats)):  
    print "#", i,"->",cats[i]  
# 0 -> garfield  
# 1 -> lola
```

# LISTS

```
cats = [ 'garfield', 'lola', 'scaredy']
print cats
# >>> ['garfield', 'lola', 'scaredy']
```

- List are mutable

```
# Mutate the value of the value at
# index 1
cats[1] = 'tom'
```

```
print cats
# >>> ['garfield', 'tom', 'scaredy']
```

## LISTS

```
name = 'Ada'  
print name[0] # A
```

```
name[0] = "B"
```

- Strings are not mutable

```
Traceback (most recent call last):  
  File "workspace.py", line 11, in  
    <module>  
      name[0] = "B"  
TypeError: 'str' object does not  
support item assignment
```

# LISTS

- A list can be empty

```
drinks = []  
print drinks # []
```

# LISTS

- Lists can be concatenated using the `+` operator

```
cats = [ 'garfield', 'lola', 'scaredy']
famous_cats = [ 'whiskers', 'grumpy
cat']

# Use the + operator to concatenate
# lists
all_cats = cats + famous_cats

print all_cats

# >>> ['garfield', 'lola', 'scaredy',
'whiskers', 'grumpy cat']
```

# LISTS

- Combined concatenation and value reassignment

```
cats = [ 'garfield', 'lola', 'scaredy']
famous_cats = [ 'whiskers', 'grumpy
cat']

# Use the += operator to concatenate
# and reassign to original list
# cats = cats + famous_cats
cats += famous_cats

print cats
# >>> ['garfield', 'lola', 'scaredy',
'whiskers', 'grumpy cat']
```

## LISTS

- Lists can be sliced
  - `list[start:stop]`

```
cats = [ 'garfield', 'lola', 'scaredy']
famous_cats = [ 'whiskers', 'grumpy
                cat']

# Use the + operator to concatenate
# lists
all_cats = cats + famous_cats

print all_cats

# >>> ['garfield', 'lola', 'scaredy',
      'whiskers', 'grumpy cat']

print all_cats[2:5]
# >>> ['scaredy', 'whiskers', 'grumpy
      cat']
```

# LISTS

- A list can be empty

```
drinks = []  
print drinks # []
```

# LISTS

- List have built-in functions
  - `append(item)`

```
drinks = []
print drinks # []

drinks.append("Soda")
print drinks # ['Soda']
```

```
drinks.append("Wine")
print drinks # ['Soda', 'Wine']

drinks.append("Beer")
print drinks # ['Soda', 'Wine', 'Beer']
```

# LISTS

- List have built-in functions
  - extend(list)

```
drinks = []
print drinks # []
```

```
drinks.append("Soda")
print drinks # ['Soda']
```

```
more_drinks = ["Wine", "Beer"]
drinks.extend(more_drinks)
print drinks # ['Soda', 'Wine', 'Beer']
```

# LISTS

- List have built-in functions
  - sort()

```
print drinks # ['Soda', 'Wine', 'Beer']

drinks.sort()

print drinks # ['Beer', 'Soda', 'Wine']
```

# LISTS

- List have built-in functions
  - sorted(list) vs list.sort()
- Pay attention to returned values of functions

```
list = [6,4,2,3]
print list
# >>> [6, 4, 2, 3]

# sorted() returns a new list the
# original list remains the same
print sorted(list)
# >>> [2, 3, 4, 6]

print list
# >>> [6, 4, 2, 3]

print list.sort()
# None

print list
# >>> [2, 3, 4, 6]
```

# LISTS

```
drinks = []
print drinks # []
```

```
drinks.append("Soda")
print drinks # ['Soda']
```

```
more_drinks = ["Wine", "Beer"]
drinks += more_drinks
print drinks
# ['Soda', 'Wine', 'Beer']
```

```
drinks.insert(0,'Lemonade')
print drinks
# ['Lemonade', 'Soda', 'Wine', 'Beer']
```

insert(index,value)

# LISTS

```
drinks = ["Soda", "Wine", "Beer", "Lemonade"]
print drinks
# ['Soda', 'Wine', 'Beer', 'Lemonade']

del drinks[0]
print drinks
# ['Wine', 'Beer', 'Lemonade']

del drinks [0:2]
print drinks
# ['Lemonade']
```

DEL LIST[INDEX]  
REMOVES THE  
ITEM AND DOES

# LISTS

```
drinks = ["Soda", "Wine", "Beer", "Lemonade"]  
del drinks[0]
```

```
print drinks  
# ['Wine', 'Beer', 'Lemonade']
```

```
removed_item = drinks.pop()  
print removed_item  
# Lemonade
```

```
print drinks  
# ['Wine', 'Beer']
```

pop returns the  
element removed

# LISTS

```
drinks = ["Soda", "Wine", "Beer", "Lemonade"]
print drinks
# ['Lemonade', 'Soda', 'Wine', 'Beer']
```

```
drinks.remove('Wine')
drinks.remove('Beer')
print drinks
# ['Lemonade', 'Soda']
```

Remove(element)  
removes the  
element

# LISTS

- dir()
  - attempt to return a list of valid attributes for that object

```
>>> x = list()
>>> type(x)

<type 'list'>
>>> y = []

>>> type(y)
<type 'list'>

>>> dir(y)
['append', 'count', 'extend', 'index',
 'insert', 'pop', 'remove', 'reverse',
 'sort']
```

## LISTS

```
>>> x = list()
>>> type(x)

<type 'list'>
>>> y = []

>>> type(y)
<type 'list'>

>>> dir(y)
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

# LISTS

`__X__` are special  
functions used  
internally by python

```
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__',
 '__delitem__', '__delslice__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getslice__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__',
 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
 '__setattr__', '__setitem__', '__setslice__', '__sizeof__',
 '__str__', '__subclasshook__', 'append', 'count', 'extend',
 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

# LISTS

- Built-in function  
`dir()`
  - Attempt to return a list of valid attributes for that object

```
>>> s = 'hi'  
>>> type(s)  
<type 'str'>  
>>> dir(s)  
['capitalize', 'center', 'count',  
'decode', 'encode', 'endswith',  
'expandtabs', 'find', 'format',  
'index', 'isalnum', 'isalpha',  
'isdigit', 'islower', 'isspace',  
'istitle', 'isupper', 'join', 'ljust',  
'lower', 'lstrip', 'partition',  
'replace', 'rfind', 'rindex', 'rjust',  
'rpartition', 'rsplit', 'rstrip',  
'split', 'splitlines', 'startswith',  
'strip', 'swapcase', 'title',  
'translate', 'upper', 'zfill']
```

# LISTS

- Lists can be function arguments

```
def add_them_up(numbers):
    # Take a list of numbers, sum
    # them and return the total
    total = 0
    for number in numbers:
        total = total + number
    return total
```

```
scores = [3, 41, 12, 9, 74, 15]
print add_them_up(scores)
```

# LISTS

- Functions can return a list

```
def first_and_last(numbers):
    # Return the first and last
    # item of a list
    first = numbers[0]
    last = numbers[-1]
    first_last = [first, last]
    return first_last
```

```
scores = [3, 41, 12, 9, 74, 15]
print first_and_last(scores)
# >>> [3, 15]
```

# LISTS

- Lists can be function arguments

```
def add_them_up(numbers):
    # Take a list of numbers, sum
    # them and return the total
    total = 0
    for number in numbers:
        total = total + number
    return total
```

```
scores = [3, 41, 12, 9, 74, 15]
print add_them_up(scores)
```

# LISTS

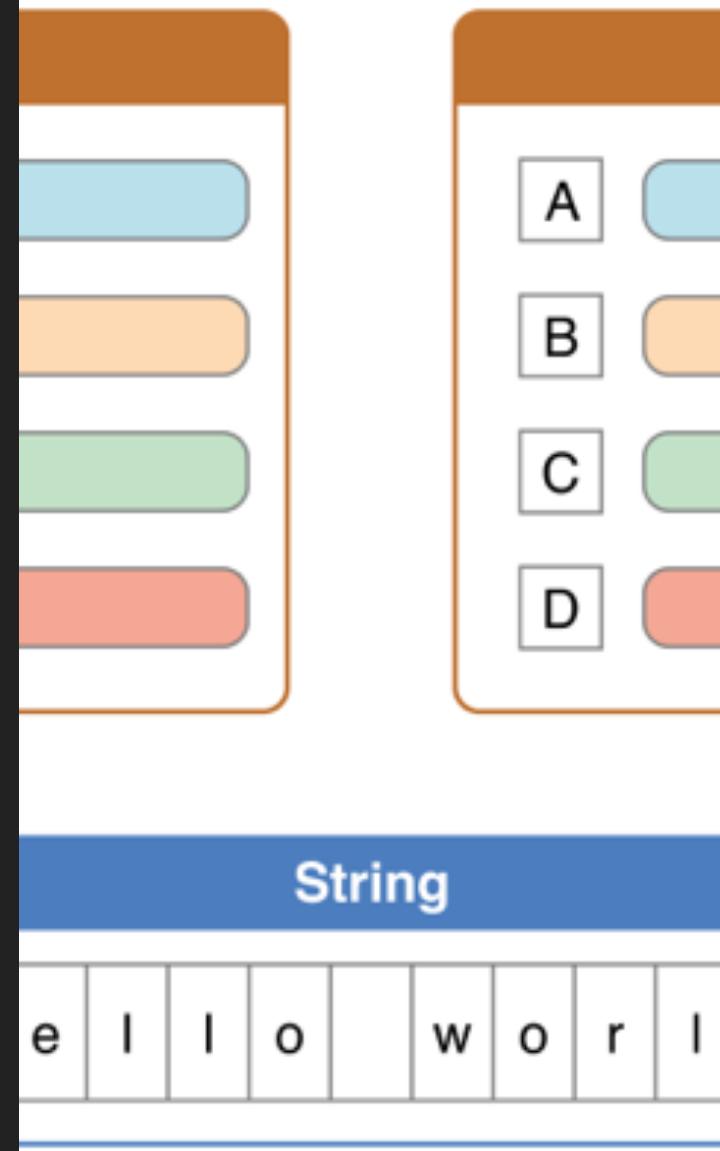
- There are a number of functions built into Python that take lists as parameters

```
nums = [3, 41, 12, 9, 74, 15]

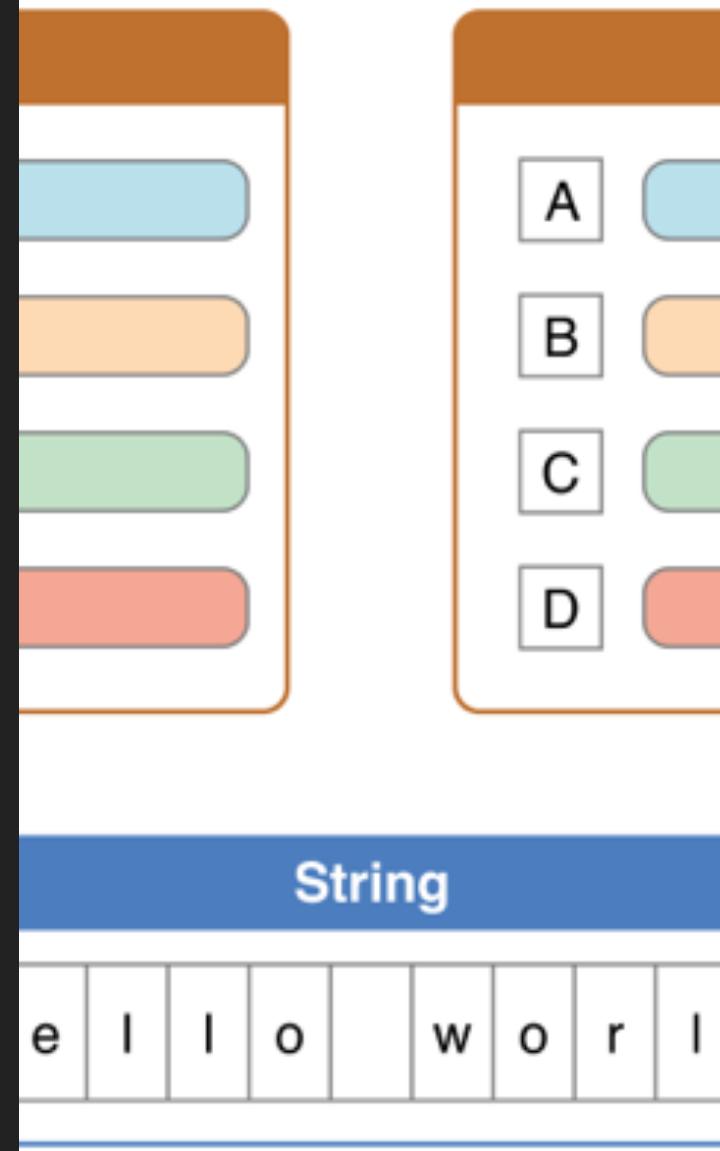
print len(nums)                  # 6
print max(nums)                 # 74
print min(nums)                  # 3
print sum(nums)                  # 154
print sum(nums)/len(nums)        # 25
```

# FOUNDATION FRAMEWORK

- ▶ Defines a base layer of functionality required for almost all applications
- ▶ Primitive classes (eg. int, float, char)
- ▶ Value classes representing basic data types (e.g. strings, numbers, collections)
- ▶ Provide a small set of basic utility classes
- ▶ Make software development easier by introducing consistent conventions



- ▶ Introduces several paradigms that define functionality not provided by either the Objective-C runtime and language or Swift standard library and language
- ▶ Support internationalization and localization, to make software accessible to users around the world
- ▶ Provide a level of OS independence, to enhance portability



• Make software development easier by introducing consistent conventions for things such as deallocation.

- Support Unicode strings, object persistence, and object distribution.
- Provide a level of OS independence, to enhance portability.

The Foundation framework includes the root object class, classes representing basic data types such as strings and byte arrays, collection classes for storing other objects, classes representing system information such as dates, and classes representing communication ports. See [Cocoa Objective-C Hierarchy for Foundation](#) for a list of those classes that make up the Foundation framework.

The Foundation framework introduces several paradigms to avoid confusion in common situations, and to introduce a level of consistency across class hierarchies. This consistency is done with some standard policies, such as that for object ownership (that is, who is responsible for disposing of objects), and with abstract classes like [NSEnumerator](#). These new paradigms reduce the number of special and exceptional cases in an API and allow you to code more efficiently by reusing the same mechanisms with various kinds of objects.

## Foundation Framework Classes

The Foundation class hierarchy is rooted in the Foundation framework's [NSObject](#) class. The remainder of the Foundation framework consists of several related groups of classes as well as a few individual classes. Many of the groups form what are called class clusters—abstract classes that work as umbrella interfaces to a versatile set of private subclasses. [NSString](#) and [NSMutableString](#), for example, act as brokers for instances of various private subclasses optimized for different kinds of storage needs. Depending on the method you use to create a string, an instance of the appropriate optimized class will be returned to you.

Many of these classes have closely related functionality:

- **Data storage.** [NSData](#) and [NSString](#) provide object-oriented storage for arrays of bytes. [NSValue](#) and [NSNumber](#) provide object-oriented storage for arrays of simple C data values. [NSArray](#), [NSDictionary](#), and [NSSet](#) provide storage for Objective-C objects of any class.
- **Text and strings.** [NSCharacterSet](#) represents various groupings of characters that are used by the [NSString](#) and [NSScanner](#) classes. The [NSString](#) classes represent text strings and provide methods for searching, combining, and comparing strings. An [NSScanner](#) object is used to [scan numbers and words from an NSString object](#).

```
components instance
    _Components(string: "https://swift.org")!
    g useful about the URL
    post!"")
    .org"
```

# NSObject

- Root class of all objects
- Implements basic functionality
  - Memory management
  - Introspection
  - Object equality

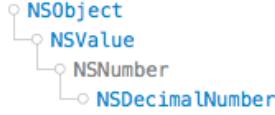
The screenshot shows the Mac Developer Library interface with the following details:

- Page Title:** NSObject Class Reference
- Page Content:** The page describes NSObject as the root class of most Objective-C class hierarchies. It lists basic interfaces like `alloc`, `copy`, and `isEqual`. It also lists methods for creating, copying, and deallocating objects.
- Left Sidebar (Tasks):** A list of tasks including: Initializing a Class, Creating, Copying, and Deallocating Objects, Identifying Classes, Testing Class Functionality, Testing Protocol Conformance, Obtaining Information About Methods, Describing Objects, Discardable Content Proxy Support, Sending Messages, Forwarding Messages, Dynamically Resolving Methods, Error Handling, Archiving, Working with Class Descriptions, Scripting, and Deprecated Methods.
- Related Sample Code:** Includes Denoise, QTCoreVideo102, QTCoreVideo103, QTCoreVideo201, and QTCoreVideo202.
- Inheritance:** Not Applicable
- Conforms To:** NSObject
- Import Statement:** `@import Foundation;`
- Availability:** Available in OS X v10.0 and later.
- Sub-sections:** Initializing a Class, Creating, Copying, and Deallocating Objects, Identifying Classes, and Testing Class Functionality.

# VALUE OBJECTS

- An object that encapsulates a primitive value (of a C data type)
- Provides services related to that value
- Some examples of value objects in the Foundation framework are:
  - `NSString` and `NSMutableString`
  - `NSData` and `NSMutableData`
  - `NSDate`
  - `NSNumber`
  - `NSValue`

# VALUE OBJECTS

<p>Related Documentation</p> <p>Number and Value Programming Topics</p> <p>Property List Programming Guide</p>	<p>Inheritance</p>  <p>NSCopying NSObject NSSecureCoding</p>	<p>Conforms To</p> <p>Import Statement</p> <pre>@import Foundation;</pre>
<p>Related Sample Code</p> <p>Advanced UISearchBar</p> <p>AdvancedURLConnections</p> <p>Core Image Filters with Photos and Video for iOS</p> <p>CryptoExercise</p> <p>TheElements</p>	<p>Creating an NSNumber Object</p> <pre>+ numberWithBool: + numberWithChar: + numberWithDouble: + numberWithFloat: + numberWithInt: + numberWithInteger: + numberWithLong: + numberWithLongLong: + numberWithShort: + numberWithUnsignedChar: + numberWithUnsignedInt: + numberWithUnsignedInteger:</pre>	<p>Availability</p> <p>Available in iOS 2.0 and later.</p>

# VALUE OBJECTS

Bytes	NSMutableCopying	Compton Foundation
Getting C Strings	NSMutableCopying	
Combining Strings	NSObject	
Dividing Strings	NSSecureCoding	
Finding Characters and Substrings		Availability
Replacing Substrings		Available in iOS 2.0 and later.
Determining Line and Paragraph Ranges		
Determining Composed Character Sequences		
Converting String Contents Into a Property List		
Identifying and Comparing Strings		
Folding Strings		
Getting a Shared Prefix		
Changing Case		
Getting Strings with Mapping		
Getting Numeric Values		
Working with Encodings		
Working with Paths		
Working with URLs		
Linguistic Tagging and		
Deleted Documentation		

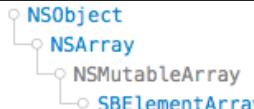
**Creating and Initializing Strings**

```
+ string  
- init  
- initWithBytes:length:encoding:  
- initWithBytesNoCopy:length:encoding:freeWhenDone:  
- initWithCharacters:length:  
- initWithCharactersNoCopy:length:freeWhenDone:  
- initWithString:  
- initWithCString:encoding:  
- initWithUTF8String:  
- initWithFormat:  
- initWithFormat:arguments:  
- initWithFormat:locale:  
- initWithFormat:locale:arguments:
```

# VALUE OBJECTS

<p>Working with String Elements Creating a Description Collecting Paths Key-Value Observing Key-Value Coding</p> <p>Constants NSBinarySearchingOptions</p> <p>Related Documentation Collections Programming Topics</p> <p>Related Sample Code From A View to A Movie From A View to A Picture QuickLookSketch Sketch Sketch+Accessibility</p>	<p>NSArray NSMutableArray</p> <p>Copying NSFastEnumeration NSMutableCopying NSObject NSSecureCoding</p> <p>Availability Available in OS X v10.0 and later.</p>
	<h3>Creating an Array</h3> <p>+ array + arrayWithArray: + arrayWithContentsOfFile: + arrayWithContentsOfURL: + arrayWithObject: + arrayWithObjects: + arrayWithObjects:count:</p> <h3>Initializing an Array</h3> <p>- init - initWithArray: - initWithArray:copyItems:</p>

# VALUE OBJECTS

<p>Related Documentation</p> <p>Collections Programming Topics</p> <p>Key-Value Coding Programming Guide</p>	 <p>NSObject NSArray NSMutableArray SBElementArray</p>	<p>NSCopying NSFastEnumeration NSMutableCopying NSObject NSSecureCoding</p> <p>@import Foundation;</p>
<p>Related Sample Code</p> <p>AVCompositionDebugViewer</p> <p>iSpend</p> <p>SimpleScriptingPlugin</p> <p>Sketch+Accessibility</p> <p>TableViewPlayground</p>		<p>Availability</p> <p>Available in OS X v10.0 and later.</p> <h3>Creating and Initializing a Mutable Array</h3> <p>+ arrayWithCapacity: + arrayWithContentsOfFile: + arrayWithContentsOfURL: - init - initWithCapacity: - initWithContentsOfFile: - initWithContentsOfURL:</p> <h3>Adding Objects</h3> <p>- addObject: - addObjectsFromArray: - insertObject:atIndex:</p>

# VALUE OBJECTS

<p>Enumerating Dictionaries Sorting Dictionaries Filtering Dictionaries Storing Dictionaries Accessing File Attributes Creating a Description</p>	<p><b>Inheritance</b></p> <pre>graph TD; NSObject[NSObject] --&gt; NSDictionary[NSDictionary]; NSDictionary --&gt; NSMutableDictionary[NSMutableDictionary]</pre>	<p><b>Conforms To</b></p> <p><a href="#">NSCopying</a> <a href="#">NSFastEnumeration</a> <a href="#">NSMutableCopying</a> <a href="#">NSObject</a> <a href="#">NSSecureCoding</a></p>	<p><b>Import Statement</b></p> <pre>@import Foundation;</pre> <p><b>Availability</b></p> <p>Available in iOS 2.0 and later.</p>
<p><b>Related Documentation</b></p> <p>Collections Programming Topics Property List Programming Guide</p>			
<p><b>Related Sample Code</b></p> <p>AdvancedURLConnections Core Image Filters with Photos and Video for iOS MVCNetworking Quartz Composer IMStatus SquareCam</p>			
<p><b>Creating a Dictionary</b></p> <pre>+ dictionary + dictionaryWithContentsOfFile: + dictionaryWithContentsOfURL: + dictionaryWithDictionary: + dictionaryWithObject:ForKey: + dictionaryWithObjects:forKeys: + dictionaryWithObjects:forKeys:count: + dictionaryWithObjectsAndKeys:</pre> <p><b>Initializing an NSDictionary Instance</b></p> <pre>- initWithContentsOfFile:</pre>			

# VALUE OBJECTS

Collections Programming Topics	<p>Inheritance</p> <pre>graph TD; NSObject --&gt; NSDictionary; NSDictionary --&gt; NSMutableDictionary</pre> <p>Conforms To</p> <p>NSObject NSDictionary NSMutableDictionary NSCopying NSFastEnumeration NSMutableCopying NSObject NSSecureCoding</p> <p>Import Statement</p> <p>@import Foundation;</p>	
Related Sample Code	<p>AdvancedURLConnections Core Image Filters with Photos and Video for iOS CryptoExercise GenericKeychain Quartz Composer IMStatus</p> <p>Creating and Initializing a Mutable Dictionary</p> <p>+ dictionaryWithCapacity: - initWithCapacity: - init + dictionaryWithSharedKeySet:</p> <p>Adding Entries to a Mutable Dictionary</p> <p>- setObject:forKey: - setObject:forKeyedSubscript: - setValue:forKey: - addEntriesFromDictionary: - setDictionary:</p>	<p>Availability</p> <p>Available in iOS 2.0 and later.</p>

# NSLOG

- ▶ Logs output to the Xcode console
- ▶ C-style type formatting for arguments
- ▶ calls `description` on an object

The screenshot shows the Xcode interface during a debug session. At the top, it says "Phone 6s Plus" and "Running TheThreeLittlePigs on iPhone 6s Plus". The "Debug" tab is selected. In the center, there's a code editor for "AppDelegate.swift" with the following content:

```
// AppDelegate.swift
// TheThreeLittlePigs
//
// Created by T. Andrew Binkowski on 1/10/16.
// Copyright © 2016 T. Andrew Binkowski. All rights reserved.

import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

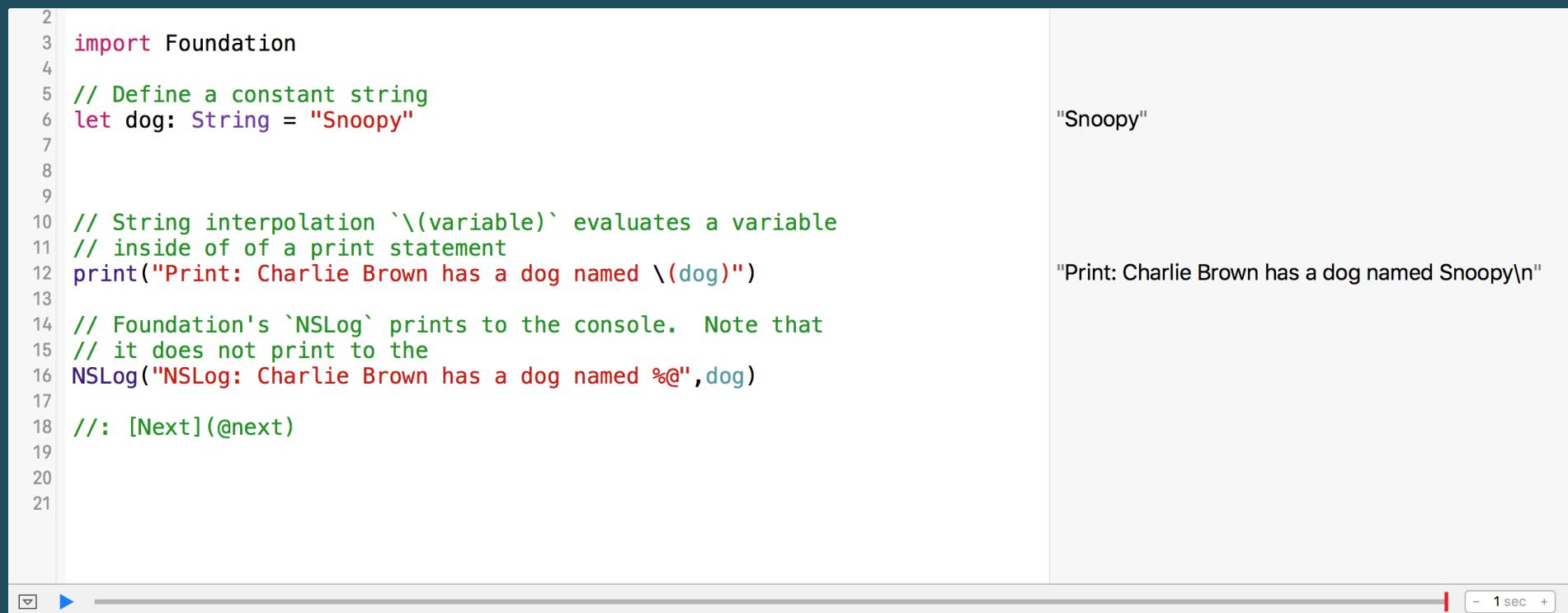
    var window: UIWindow?

    func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) {
        // Override point for customization after application launch.
        NSLog("The application is launching")
    }
}
```

Below the code editor, the Xcode toolbar is visible. The bottom part of the screen shows the "Output" tab, which displays the log message:

```
1-11 10:26:29.752 TheThreeLittlePigs[20187:1842429] The application is launching
```

# NSLOG



The screenshot shows a code editor with two columns. The left column contains Objective-C code demonstrating string interpolation and NSLog. The right column shows the resulting console output.

```
2 import Foundation
3
4 // Define a constant string
5 let dog: String = "Snoopy"
6
7
8
9
10 // String interpolation `\\(variable)` evaluates a variable
11 // inside of of a print statement
12 print("Print: Charlie Brown has a dog named \\(dog)")
13
14 // Foundation's `NSLog` prints to the console. Note that
15 // it does not print to the
16 NSLog("NSLog: Charlie Brown has a dog named %@", dog)
17
18 //: [Next](@next)
19
20
21
```

"Snoopy"

"Print: Charlie Brown has a dog named Snoopy\n"

- Print vs. NSLog

# WHAT ABOUT SWIFT TYPES

- ▶ Swift has native types
- ▶ Compatible with Foundation counterparts
- ▶ Some have same/more/less functionality

n

work defines a base layer of functionality that is required for almost all applications. It introduces several paradigms that define functionality not provided by either the standard library and language.

goals in mind:

of basic utility classes.

development easier by introducing consistent conventions.

ization and localization, to make software accessible to users around the world.

independence, to enhance portability.

on the Foundation framework [here](#).

This is in the early stages of development. It is not yet ready for production use. It is scheduled to be part of the Swift 3 release.

The implementation of the Foundation API for platforms where there is no Objective-C API is still in progress. On non-Apple platforms, apps should use the Foundation that comes with the operating system, rather than the underlying platform as much as possible.

We are also working to achieve implementation parity with Foundation on Apple platforms. This will happen over time as we add new types.

We are not looking to make major API changes to the library. We feel that this will be avoided where API changes are unavoidable, however. For more information on those changes, please see [our design document](#).

# WHAT ABOUT SWIFT TYPES

```
Foundation.xcplaygroundpage
Session2-SwiftBasics > Foundation
10 let dog: String = "Snoopy"
11
12 // String interpolation `\"(variable)` evaluates a variable
13 // inside of of a print statement
14 print("Print: Charlie Brown has a dog named \(dog)")
15
16 // Foundation's `NSLog` prints to the console. Note that
17 // it does not print to the
18 NSLog("NSLog: Charlie Brown has a dog named %@", dog)
19
20
21 //: Swift vs. Foundation
22 //: -----
23 var greeting = "Hello from Swift"
24
25 // Make it more exciting using an exclusive Swift `String` API
26 greeting.appendContentsOf("!")
27
28 // Make it more exciting for 2016 using a Foundation NSString API
29 greeting.stringByReplacingOccurrencesOfString("!", withString: "😊")
30
31 // Make it more exciting for the snapchat generation
32 let image = UIImage(named:"swift.jpg")
33
34
35 // - Swift 2.0 will bring features parity
```

Snoopy  
"Print: Charlie Brown has a dog na..."  
  
"Hello from Swift"  
  
"Hello from Swift!"  
  
"Hello from Swift😊"  
  
w 620 h 506

# WHAT ABOUT SWIFT TYPES

- ▶ Swift 3
- ▶ Rewrite Foundation for portability
- ▶ Simplify naming (no more NS)
- ▶ Feature parity

h

rk defines a base layer of functionality that is required for almost all applications. It introduces several paradigms that define functionality not provided by either the standard library and language.

goals in mind:

of basic utility classes.

development easier by introducing consistent conventions.

ization and localization, to make software accessible to users around the world.

independence, to enhance portability.

on the Foundation framework [here](#).

is in the early stages of development. It is not yet ready for production use and is scheduled to be part of the Swift 3 release.

implementation of the Foundation API for platforms where there is no Objective-C Foundation API. On non-Apple platforms, apps should use the Foundation that comes with the operating system and interact with the underlying platform as much as possible.

achieve implementation parity with Foundation on Apple platforms. This will happen over time.

not looking to make major API changes to the library. We feel that this will be the case. Where API changes are unavoidable, however. For more information on those changes see [our design document](#).