Swift Programming

iOS 10 • Xcode 8

STUDENT GUIDE

ABOUTOBJECTS

Contact

About Objects
11911 Freedom Drive
Suite 700
Reston, VA 20190

main: 571–346–7544
email: info@aboutobjects.com
web: www.aboutobjects.com

Course Information

Author: Jonathan Lehr
Revision: 4.1.0
Last Update: 8/25/2017

Classroom materials for an course that provides a rapid introduction to programming in Swift. Geared to developers interested in learning to do Cocoa development on the iOS platform. Includes comprehensive lab exercise instructions and solution code.

Copyright Notice

Swift Programming

STUDENT GUIDE

About the Instructor

Jonathan Lehr

Co-Founder and VP, Training

About Objects

jonathan@aboutobjects.com

ABOUTOBJECTS

CHAPTER ONE

Swift Basics

About Swift

- Why Swift?
  - Safety
  - Speed
  - Modern language features (functional programming, closures, pattern matching, tuples, optionals, etc.)
- Apple plans to use Swift for:
  1. Systems programming
  2. Scripting
  3. Cocoa development (replacing Objective-C)
- Existing Cocoa frameworks are all C/Objective-C
  - Swift designed to allow easy bridging between languages
  - Uses Objective-C runtime under the hood

Everything's an Object

There are four object types in Swift:

- Class
- Struct
- Enum
- Protocols

Even literals — for example the numeric literal `42` — are objects (in this case, an instance of a struct).

```
42.advanced(by: 7)   // returns 49
```

You can even customize the behavior of fundamental types:

```
extension Int
{
    func printAsAge()
    {
        print("I'm \(self) years old")
    }
}

// Adds printAsAge method to the Int type

42.printAsAge()

// prints "I'm 42 years old"
```

Variable Declarations

Variable declarations must provide either explicit type information, or an explicit initial value.

```
var width: Int   // defines variable 'width' of type Int

var height = 12  // defines 'height' with inferred type Int
```

Note that the compiler will trap any attempts to use a variable before it has been initialized, as shown in the following example:

```
var count: Int
count += 1    // Compiler error.
```

You can declare several variables of the same type in a single statement:

```
var x, y, z: Int
var i = 0, j = 0
```

Constant Declarations

Constants declarations are similar to variable declarations, except they use the keyword `let` instead of `var`. However, constants must always be declared with an initial value.

```
let width: Double = 8.0  // declares constant of type Double
let height = 5.0  // defines constant with inferred type Double
```

The compiler will trap any attempt to modify the value of a constant.

```
let pi = 3.14159
pi += 2.0           // Compiler error.
```

Numeric Types

- Swift Standard Library declares protocols such as **Integer**, **SignedInteger**, etc. that define requirements for integer types.
- Protocols used in declaring structs to represent various integer types, such as **Int**, **Int8**, **UInt8**, **Int16**, **UInt16**, etc.
- Similarly, protocol **FloatingPoint** is adopted by both **Float** and **Double**.
- You can use constructors to convert between dissimilar types.

```
let x = 42
let y: Float = x          // Illegal!
let z: Float = Float(x)   // This works fine
let easier = Float(x)     // This works fine too
let backToInt = Int(z)    // Yep
```

Printing Text and Values

The Swift Standard Library provides `print` function for printing text and interpolated values:

```
print("Have a nice day")
// prints "Have a nice day"
```

`print` is a generic function that takes an argument of any type:

```
let message = "Have a nice day"
print(message)
// prints "Have a nice day"
let pi = 3.14159
print(pi)
// prints "3.14159"
```

Values of any type can be interpolated in a string literal with `\()`.

```
let temperature = 72
print("It's currently \(temperature) degrees")
// prints "It's currently 72 degrees"

let scale = "Fahrenheit"
print("It's currently \(temperature) degrees \(scale)")
// prints "It's currently 72 degrees Fahrenheit"
```

Strings and Characters

- Owns a collection of **Character**
  - Each character represents a Unicode character.
  - Width of Unicode characters can vary, therefore indexes of characters are computed by enumerating the string.

```
let emojiText = "Hello World! 😊🌍"
emojiText.characters.count           // returns 15

let foundationEmojiText: NSString = emojiText as NSString
foundationEmojiText.length            // returns 17
```

- Working with String properties:

```
let text = "Hello World!"
text.isEmpty              // prints false
text.lowercased()         // prints "hello world!"
text.uppercased()         // prints "HELLO WORLD!"
text.hasPrefix("Hello")   // prints true
text.hasSuffix("World")   // prints false
```

Bridging to Foundation

Objective-C

- Swift interoperates with C and Objective-C code
- Swift currently depends on Objective-C runtime

Foundation Framework

- Objective-C classes, structures, enums, and C functions
- All global symbols prefixed with `NS`
- Some Swift types (e.g. `String` and `Array`) are bridged to corresponding Foundation classes (e.g., `NSString` and `NSArray`)
- Swift 3 requires dynamic casts to access bridged types

```
let word = "Hello" as NSString
```

String Formatting

- Strings can be initialized with a `printf`-style format string and variable length argument list (variadic):

- Arguments from variadic list are interpolated in place of *format specifiers*.

```
let foo: NSString = "Foo"
// Uses %d format specifier for Int length
let s1 = String(format: "foo's length is %d", foo.length)
// prints "foo's length is 3"


let fahrenheit = 78.5
// Uses %.1f format specifier for Double value, where the '.1'
// specifies one digit of decimal precision.
let s2 = String(format: "temperature is %.1f°F", fahrenheit)
// prints "temperature is 78.5°F"
```

- See Apple's documentation for a comprehensive list of format specifiers.

Functions and Methods

- Function and method declarations share similar syntax:

```
// function declaration syntax
func display(degrees: Double, scale: String)
{
    print("The temperature is \(degrees)° \(scale)")
}

// function call syntax
display(degrees: 71.5, scale: "Fahrenheit")
// prints "The temperature is 71.5° Fahrenheit"
```

- A parameter can declare a separate *external* name:

```
// temperatureInDegrees is external name of first param
func display(temperatureInDegrees degrees: Double, scale: String)
{
    // internal name is degrees
    print("The temperature is \(degrees)° \(scale)")
}

// external name required in call
display(temperatureInDegrees: 71.5, scale: "Fahrenheit")
```

- A parameter's external name can be ignored:

```
// external name ignored if _ wildcard specified
func display(_ degrees: Double, scale: String)
{
    print("The temperature is \(degrees)° \(scale)")
}

// external name not permitted in call
display(71.5, scale: "Fahrenheit")
```

Return Values

- Return type declared after **->** symbol:

```
// returns a value of type String
func temperature(degrees: Double, scale: String) -> String
{
    return "The temperature is \(degrees)° \(scale)"
}
```

- Result of function call must be used in an expression

```
// compiles with warning ("Result of call '...' is unused")
temperature(degrees: 71.5, scale: "Fahrenheit")

// compiles cleanly
let s = temperature(degrees: 71.5, scale: "Fahrenheit")
```

- Declare with `@discardableResult` to allow return value to be ignored

```
// return value can be ignored

@discardableResult
func temperature(degrees: Double, scale: String) -> String
{
    let s = "The temperature is \(degrees)° \(scale)"
    print(s)
    return s
}
```


Lab 1: Temperature Conversion

1. Write a function to convert Fahrenheit to Celsius
    1.1. Subtract 32 and multiply by 5/9.
    1.2. Write a unit test that passes in several different values and prints the results.
2. Write a function to convert Celsius to Fahrenheit
    2.1. The algorithm is the inverse of the one used in Step 1.
    2.2. Write a unit test that passes in several different values and prints the results.

Generics

- Swift provides rich support for generic types.
  - Used heavily in the standard library.
- The example below uses the `Comparable` protocol as a type qualifier. We haven't introduced protocols yet, but they're similar to *interfaces* in other languages.

```
// defines a function that takes two Comparable values
// and returns the larger of the two
//
func maxValue<T: Comparable>(x: T, y: T) -> T
{
    return x > y ? x : y
}

maxValue(22.5, 23)           // returns 23
maxValue(3, 2.9)             // returns 3
maxValue("Apple", "Banana")  // returns "Banana"
```

- The *T* above is a *placeholder type*. Here it denotes that the args and the return value must be the same type.
- `<T: Comparable>` is a *type constraint.* Here it specifies that `T` matches only types that conform to `Comparable`.

Collections

- Collections are generically typed.
- Three primary collection types:

  arrays: ordered values

  sets: unordered, unique values

  dictionaries: unordered key-value pairs; keys are unique
- Collections defined with `let` are immutable.

Arrays

- Declaring an array

```
// Declare an array with generic type parameter
var a: Array<Int>
// Declare an array using shorthand syntax (preferred)
var a: [Int]
```

- Defining an instance

```
// Instantiating an array
var a = Array<Int>()
// ...using shorthand syntax
var a = [Int]()
// ...without type inference
var a: [Int] = []
```

- Array literal syntax

```
var words = ["one", "two", "three"]
print(words)
// ["one", "two", "three"]
```

- Accessing elements by index

```
print(words[1])
// "two"
words[0] = "uno"
// ["uno", "two", "three"]
```


Array API Basics

- Accessing array properties:

```
var words = ["one", "two", "three"]
// ["one", "two", "three"]
print(words.count)
// 3
print(words.first)
// "one"
```

- Using operators:

```
// Note: + operator defined for RangeReplaceableCollection type
var words2 = words + ["four", "five"]
// ["one", "two", "three", "four", "five"]
```

- Inserting and removing elements:

```
words.insert("ONE", at: 0)
// ["ONE", "uno", "two", "three"]
words.remove(at: 1)
// ["ONE", "two", "three"]
words.append("four")
// ["ONE", "two", "three", "four"]
words.append(contentsOf: ["five", "six"])
// ["ONE", "two", "three", "four", "five", "six"]
```

- Manipulating elements:

```
print(words.joined(separator: ", "))
// ONE, two, three, four, five, six

print(words.sorted())
// ["ONE", "five", "four", "six", "three", "two"]
```

Dictionaries

- Declaring a dictionary

```
// declaring with keys of type String and elements of type Any
var a: Dictionary<String, Any>
// ...using shorthand syntax
var b: [String: Any]
```

- Creating an instance

```
// using generic type syntax
var a = Dictionary<String, Any>()
// ...using shorthand syntax
var a = [String: Any]()
// ...without type inference
var a: [String: Any] = [:]
```

- Working with elements

```
// accessing
// (note: resulting value wrapped in an Optional)
let age = a["age"]

// inserting/modifying elements
a["name"] = "Fred"
a["age"] = 29
```

- Dictionary literals

```
// type can be inferred for homogenous elements
let c = ["min": 0, "max": 99, "average": 42.5]

// type annotation required for mixed elements
let d: [String: Any] = ["name": "Fred", "age": 29 ]
```


For Loops

- Looping through an array:

```
let names = ["Jane", "Bill", "Jan"]

for name in names {
    print("name is \(name)")
}
// name is Jane
// name is Bill
// name is Jan
```

- Looping through a dictionary:

```
let prices = ["jeans": 49.99, "t-shirt": 29.99]

for (key, value) in prices {
    print("price of \(key) is \(value)")
}
// price of jeans is 49.99
// price of t-shirt is 29.99
```


enumerated Method

- `enumerated` method sequences through a collection's elements.
  - Returns a tuple containing the index of the current element and the value at that index.

```
let names = [ "Jane", "Bill", "Jan", "Pat" ]
// defines an array of elements of type String

for (index, value) in names.enumerated()
{
    print("name \(index + 1) is \(value)")
}
// enumerates the 'names' array, printing the following:
// name 1 is Jane
// name 2 is Bill
// name 3 is Jan
// name 4 is Pat
```

Lab 2: Collections

Write unit tests to experiment with collections.

1. Add a new file **CollectionsLabTests** as follows:

    1.1. From Xcode's **File** menu select **New -> File.** In the Template Chooser, select **iOS** at the top, select the **Unit Test Case Class** template. Click **Next,** enter **CollectionsLabTests** as the file name, and click **Next.** In the Save panel, click **Create**.

    1.2. Write a test method named **testArray** that initializes a variable with with an empty array of type String. Add code that uses the array's **append** method to append "Apple" and "Pear" to the array. Add a call to the **print** function to print the array, and then run the test to verify that the array prints as expected.

    1.3. Add a line of code to change the value of the array's second element from "Pear" to "Orange" and another line to print the array, and then run the test again.

2. Add a test method named **testEnumerateArray** that defines a let constant initialized with an array literal containing the strings "Apple" and "Banana".

    2.1. Add a **for** loop that prints each element of the array, then run the test to verify that the output is as expected.

    2.2. Add another **for** loop that uses the array's **enumerate** method to provide a tuple of each element's index and value, and then add a line of code in the loop body that prints the current index and value.

3. Add a test method named **testEnumerateDictionary** that defines a variable initialized with an empty dictionary literal.

    3.1. Use subscript notation to insert two key-value pairs with the following keys and values: "jeans", 49.99 and "t-shirt", 29.99.

    3.2. Add a line of code to print the dictionary, then run the test to verify the result.

4. Add a test method named **testEnumerateDictionary2** that defines a let constant initialized with a dictionary literal containing the same values as in the previous exercise.

    4.1. Add a **for** loop that enumerates the dictionary, printing its keys and values, followed by another **for** loop that prints only the dictionary's keys, and a third **for** loop that sums the dictionary's values. Add a print statement after the third loop that prints the sum.

Tuples

- A *tuple* is a list of objects of any type.
  - Defined as a parenthesized list of *values*.
  - Declared as a parenthesized list of *types*.

```
let vals = (12, "Hi")
// defines a tuple with two values

let typedVals: (Int, String) = (12, "Hi")
// illustrates types inferred by compiler in previous example
```

- Use the dot operator to access elements by position:

```
print(vals.0)   // prints "12"
print(vals.1)   // prints "Hi"
```

- The dot operator can also access values by label:

```
let labeledVals = (x: 12, y: "Hi")
print(labeledVals.x)  // prints "12"
print(labeledVals.y)  // prints "Hi"
```


Tuples As Types

- You can use tuples in type declarations, for example as a function parameter or return value.

```
// takes a single argument, size, of type (Double, Double)
func area(size: (Double, Double)) -> Double
{
    return size.width * size.height
}

// returns (Double, Double)
func discounted(_ price: Double, _ discount: Double) -> (Double, Double)
{
    let amount = price * discount
    return (price - amount, amount)
}

// defines a tuple, vals
let vals = discounted(25.00, 0.15)
print(vals.0)           // prints "21.25"
print(vals.1)           // prints "3.75"

// defines two separate let constants, price and discount
let (price, discount) = discounted(25.00, 0.15)
print(price)            // prints "21.25"
print(discount)         // prints "3.75"

// defines a single let constant, price, and ignores the second value
let (discPrice, _) = discounted(25.00, 0.15)
print(discPrice)        // prints "21.25"
```

Typealias

- A *typealias* is a custom label for an existing type

Example

- Here's a rewritten definition of **area** (see previous page) with labeled tuple values; it's a bit more difficult to read

```
// Tuple parameter is now more difficult to read
func area(size: (width: Double, height: Double)) -> Double
{
    return size.width * size.height
}
```

- You can declare a typealias that will allow you to simplify the function prototype, making it easier to read

```
// Declares a new type alias named Size
typealias Size = (width: Double, height: Double)
```

- You can now use *Size* as the parameter type

```
// Using the new typealias cleans up the mess
func area(size: Size) -> Double
{
    return size.width * size.height
}
```


Lab 3: Tuples

Write unit tests to experiment with tuples.

1. Add a new test case named **TuplesLabTests**.
2. Add a test method named **testTuplePositions** that defines a let constant named **item**, initialized with a tuple containing following values: "polos", 29.99, and 2. Define another let constant named **amount**, initialized with the result of an expression that multiplies the **item** tuple's second and third values. Write a print statement that prints each of the tuple's values followed by the calculated amount.
3. Define a global let constant named **polos** with values from the tuple in the previous step, prefixed with the following labels: **name**, **price**, and **quantity**.
    - 3.1. Add a test method named **testTupleLabels** that defines a let constant named **amount**, initialized with an expression that multiplies the polo constant's **price** times its **quantity**.
    - 3.2. Write a print statement that prints the **polos** tuple's values, followed by the calculated amount.
4. Write a global function named **calculatedAmount(item:)** that takes a tuple of **String**, **Double**, and **Int** as its argument, and returns **Double**. It should return the product of the provided tuple's price and quantity. Add a unit test named **testTupleParameter** that calls the new function, passing **polos** as its argument, and prints the result in the same manner as in the previous exercise.
5. Write a global function named **formatted(item:)** that takes a parameter of the same type as in the previous exercise, and returns a tuple of **String** and **Double**.
    - 5.1. The function should call **calculatedAmount**, and then return a tuple containing a string similar to the one printed in **3.2**, and the amount.
    - 5.2. Add a global let constant named **shirts**, initialized with a literal array of two tuples similar to the one in **polos**.
    - 5.3. Add a test method named **testTupleReturnValue** that calls the new **formatted(item:)** function once for each of the elements of the **shirts** array. It should print the text in each of the returned tuples and total their amounts, printing the total.

Control Flow

- if, guard, for, do, switch
- precondition, assert, fatalError

Enums

- **Enum** is a fundamental Swift type
- Declared with `enum` keyword

```
enum Garment {
    case tie
    case shirt
    case pants
}
```

- Often used with switch statements

```
func showSpecials(garmentType: Garment) {
    switch garmentType {
    case .shirt:
        print("All shirts 15% off this week.")
    case .pants:
        print("Get two pairs for the price of one!")
    default: break
    }
}
```

Associated Values

- Any enum case can be declared as having one or more associated values of any type

```
enum Garment {
    case tie
    case shirt(size: String)
    case pants(waist: Int, inseam: Int)
}
```

- Enum instances must be initialized with the specified numbers and types of associated values

```
let coolShirt = Garment.shirt(size: "XL")
let nicePants = Garment.pants(waist: 32, inseam: 34)
```

Unwrapping Associated Values

- Use `case let` constructs in switch statements to unwrap associated values:

```
enum Garment: CustomStringConvertible {
    case tie
    case shirt(size: String)
    case pants(waist: Int, inseam: Int)

    // Unwraps associated values to fully describe instance
    var description: String {
        switch self {
        case .tie:              return "tie"
        case let .shirt(s):     return "shirt: \(s)"
        case let .pants(w, i): return "pants: \(w)X\(i)"
        }
    }
}

// Defines an array of items
let items: [Garment] = [.tie,
                        .shirt(size: "XL"),
                        .pants(waist: 32, inseam: 34)]
for item in items {
    print(item) // Accesses description property
}
// tie
// shirt, XL
// pants, 32X34
```

Guard Statements

- Guard statements can be used with ordinary logic expressions:

```
func divide1(numerator: Int, denominator: Int) -> Int?
{
    guard denominator != 0 else {
        print("Zero divide")
        return nil
    }

    return numerator / denominator
}
```

- Control flow keywords such as **if**, **guard**, and **for** can also be used in combination with the **case** keyword:

```
func showDiscount(type: Garment) {
    guard case .shirt = type else {
        return
    }
    print("15% discount")
}
```

For Loop Conditionals

- Adding a `where` clause to a for loop:

```
for index in 0...5 where index % 2 == 0 {
    print("index is \(index)")
}
// index is 0
// index is 2
// index is 4
```

Other Uses of Case Let

- `case let` can be used in logic expressions.

`if` statement:

```
let supplies = [("pens", 1.75), ("pads", 1.25), ("glue", 2.50)]
let item = supplies[0]

if case let (name, price) = item, price < 2 {
    print("\(name): $\(price)")
}
// pens: $1.75
```

`for` loop:

```
for case let (name, price) in supplies where price < 2 {
    print("\(name): $\(price)")
}
// pens: $1.75
// pads: $1.25

// The compiler infers case let in for loops,
// so the previous code can be simplified:
//
for (name, price) in supplies where price < 2 {
    print("\(name): $\(price)")
}
```

CHAPTER TWO

Swift Types

Structs

- Value types — can be allocated and passed by value; i.e., copied on stack.
- Declared with **struct** keyword, followed by curly braces
- Can contain properties, methods, and initializers (constructors)

```
struct Dog {
    var name = "Unknown"
    func bark() {
        print("Woof, woof!")
    }
}

// Declares Dog to be a struct with a 'name' property and 'bark' method.
```

```
var rover = Dog()     // allocates a Dog instance
rover.name = "Rover"  // modifies the name property (conceptually;
                      // actually, replaces rover with a new instance)
```

```
print("Name of \(Dog.self) is \(rover.name)")

// prints "Name of MyProj.Dog is Rover". ('MyProj' is name of Swift module.)

rover.bark()

// prints "Woof, woof!"
```


Stored Properties

- Value stored internally in struct, class, or enum instance.
  - **Properties only declare accessors**; actual storage is opaque.
- Compiler requires all stored properties to be initialized.
  - Properties that aren't initialized by default values *must* have their values set by initializer methods.

```
struct Point {
    var x = 0.0
    var y = 0.0
}
// declares a struct whose properties all have default values
```

  - Swift synthesizes a *memberwise initializer* for all read-write properties.

```
let p2 = Point(x: 10.0, y: 20.0)
// calls memberwise initializer, returning point with origin
// 10.0, 20.0
```

  - If all properties have default values and there's no custom initializer, *default (no-arg) initializer* is synthesized.

```
let p = Point()
// calls default initializer, returning point with origin 0.0, 0.0
```


Custom Initializers

- Default values can also be provided in an `init` method's parameter list.
  - Parameters that have default values are not required in calls to initializers.

```
struct Point
{
    var x = 0.0
    var y = 0.0

    let pointsPerPixel: Double

    // pointsPerPixel not required in calls to initializer.
    // If omitted, default value (2.0, in this case) is used.
    init(x: Double, y: Double, pointsPerPixel: Double = 2.0)
    {
        self.x = x
        self.y = y
        self.pointsPerPixel = pointsPerPixel
    }
}

let p1 = Point(x: 10.0, y: 20.0)
let p2 = Point(x: 10.0, y: 20.0, pointsPerPixel: 3.0)
// Both the above calls succeed
```

Computed Properties

- No stored value
- Value computed each time property accessed

Defining a Read-Only Computed Property

```
struct Dog
{
    var name: String
    var toys: [String]

    // read-only computed property
    var favoriteToy: String {
        get { return toys.isEmpty ? "N/A" : toys[0] }
    }
    // ...
}

var rover = Dog(name: "Rover", toys: ["Ball", "Rope", "Frisbee"])

print("favorite toy: \(rover.favoriteToy)")

// prints "favorite toy: Ball"
```

- Keyword **get** may be omitted when defining read-only property

Simplified Definition of Read-Only Computed Property

```
var favoriteToy: String {
    return toys.isEmpty ? "N/A" : toys[0]
}
```


Read-Write Computed Properties

- Must provide both getter and setter

Adding a Read-Write Computed Property to Dog

```
struct Dog
{
    // ...
    var toyNames: String {
        get { return toys.joined(separator: ", ") }
        set { toys = newValue.components(separatedBy: ", ") }
    }
    // ...
}

rover.toyNames = "Kong, Ball, KittyKat"

print("toys: \(rover.toys)")

// prints "toys: [Kong, Ball, KittyKat]"
```

Property Observers

- Can be used with anything declared as `var`, as long as it has an explicit type and initial value.
  - `willSet` automatically passed `newValue`
  - `didSet` automatically passed `oldValue`

Defining a `didSet` Property Observer

```
var name: String = "Unknown" {
    didSet {
        print("Set name to \(name), old value was \(oldValue)")
    }
}

rover.name = "Rover"
// prints "Set name to Rover, old value was Unknown"
```

Defining `didSet` and `willSet` Property Observers

```
var name: String = "Unknown"  {
    willSet {
        print("About to set \(name) to new value \(newValue)")
    }
    didSet {
        print("Name is now \(name); old value was \(oldValue)")
    }
}

rover.name = "Rover"
// About to set Unknown to new value Rover
// Name is now Rover; old value was Unknown
```

Methods

- Syntax the same as for functions

```
// Defines bark method
func bark(prefix: String, suffix: String, numberOfTimes: Int)
{
    for _ in 1...numberOfTimes {
        print("\(prefix), \(barkText) \(suffix)")
    }
}

// Invokes bark method
fido.bark(prefix: "Grrr", suffix: "(pant, pant)", numberOfTimes: 2)
```

- A parameter's *external* name can be different from its *internal* name.

```
// The third parameter's external name is repetitions,
// but its internal name is numberOfTimes
func bark(prefix: String, suffix: String, repetitions numberOfTimes: Int)
{
    // ...
}

fido.bark(prefix: "Grrr", suffix: "(pant, pant)", repetitions: 2)
```

Protocols

- Provide a means to share method and property *declarations* among structs, classes of various types
- Similar to **interfaces** in other languages, but allows methods to be declared as optional *(NOTE: Swift 2 protocol extensions can also provide* implementations *of protocol methods)*
- Declared with **protocol** keyword

```
protocol Likeable {

    var numberOfLikes: Int { get set } // property declaration

    // method declarations
    func like()
    func unlike()
}
```

```
class Person: Likeable {
    // ...
    var numberOfLikes = 0
    // ...

    func like() {
        numberOfLikes += 1;
    }

    func unlike() {
        if (numberOfLikes > 0) {
            numberOfLikes -= 1
        }
    }
```

Describing Objects

- `CustomStringConvertible` protocol declares ***`description`*** property
  - Called by `print` function
- `CustomDebugStringConvertible` protocol declares ***`debugDescription`*** property
  - Used by debugger as part of formatted value presented by its built-in `print-object` (`po`) command

```
class Person: Likeable, CustomStringConvertible
{
    ...
    var description: String {
        return "\(firstName) \(lastName), +\(numberOfLikes)"
    }
}
```

```
let fred = Person(firstName: "Fred", lastName: "Smith")

fred.like()
print(fred)
// Fred Smith, +1

fred.like()
print(fred)
// Fred Smith, +2
```


The Equatable Protocol

- Declares a generic function that defines the behavior of the == operator
- Can be overloaded for custom types

```
// From declaration in Swift Standard Library:

protocol Equatable {
    func ==(lhs: Self, rhs: Self) -> Bool
}
```

```
// Overloading for Friendable type

func ==(lhs: Friendable, rhs: Friendable) -> Bool
{
    return lhs.friendID == rhs.friendID
}
```

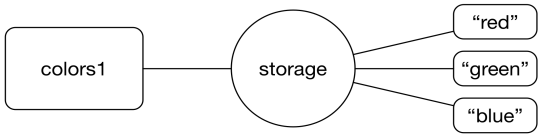
```
func testEquatable() {
    let p1 = Person("Fred", "Smith", 100)
    let p2 = Person("Fred", "Smith", 100)
    let p3 = Person("Fred", "Smith", 99)

    print("p1 == p1 is \(p1 == p1)")
    // p1 == p1 is true
    print("p1 == p2 is \(p1 == p2)")
    // p1 == p2 is true
    print("p1 == p3 is \(p1 == p3)")
    // p1 == p3 is false

    print("p1 === p1 is \(p1 === p1)")
    // p1 === p1 is true
    print("p1 === p2 is \(p1 === p2)")
    // p1 === p2 is false
}
```

Collections and Mutability

- Swift collections are structs. Each instance contains a nested class instance (i.e., reference type) that provides storage for the collection's elements.

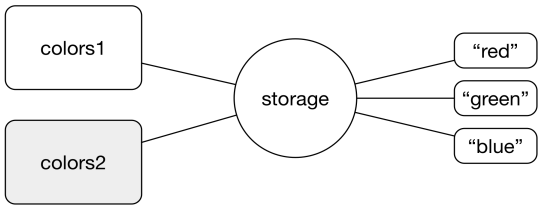
- Collections are conceptually immutable.

```
let colors1 = ["red", "green", "blue"]
```



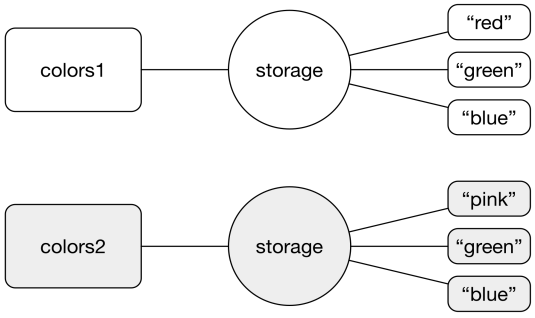
  - When a collection's struct is copied on the stack, the copy shares a reference to the original struct's elements.

```
var colors2 = colors1
```



  - If an attempt is made to mutate the copy, its nested storage is copied on write.

```
colors2[0] = "pink"
```



Lab 4: Rectangle

1. Declare three structs, as follows:
    1.1. A **Point** structure with properties **x** and **y** of type Double.
    1.2. A **Size** structure with properties **width** and **height** of type Double.
    1.3. A **Rectangle** structure with properties **origin** and **size** of type **Point** and **Size**, respectively.
2. Add a computed property in **Rectangle** that returns the area of the rectangle as a Double.
3. Add another computed property in **Rectangle** that returns a **Point** that represents the rectangle's center.
4. Add a method to **Rectangle** that takes two parameters, **dx**, and **dy**, and returns a rectangle instance offset from the original by the amount of the arguments.
5. Write unit tests to test the functionality developed in the previous steps.

Extensions

- Add methods and read-only properties to existing structs, classes, protocols, and enums
- Declared with **extension** keyword, followed by the name of the type being extended, followed by curly braces
  - Can specify additional protocols

```
extension Dog {

    var numberOfLegs: Int { return 4 } // Computed property (read-only)

    func howl() {
        print("Awooooooh!")
    }

    func growl() {
        print("Grrrrrr!")
    }
}
// Adds 'howl' and 'growl' methods to the Dog class
```

```
var fido = Dog()

fido.howl()                 // prints "Awooooooh!"
fido.growl()                // prints "Grrrrrr!"
print(rover.numberOfLegs)   // prints "4"
```

String Methods

- Joining and splitting:

```
let fruits = ["Apple", "Pear", "Banana"]
let fruitText = fruits.joined(separator: ", ")
print(fruitText)
// prints "Apple, Pear, Banana"

let fruitArray = fruitText.components(separatedBy: ", ")
print(fruitArray)
}
// prints "[Apple, Pear, Banana]"

let text = "Hello World!"
text.characters.count
//prints 12
```

String Ranges

- A string's characters can be enumerated like an array:

```
for currChar in text.characters {
    print(currChar)
}
// prints each character in the string, one per line
```

- Use ranges to specify portions of a string:

```
let startIndex = text.startIndex.advancedBy(6)
text[startIndex] // returns "W"

let endIndex = text.endIndex.advancedBy(-1)
text[endIndex]   // returns "!"

let range1 = Range(start: startIndex, end: endIndex)
text[range1]     // returns "World"

// You can use a Range literal to specify a range:
//
let range2 = startIndex ..< endIndex  // Half-open range
text[range2]     // returns "World"

let range3 = startIndex ... endIndex  // Closed range
text[range3]     // returns "World!"
```


Bridging to Foundation

- Many Swift library structs such as `String`, `Array`, `Dictionary`, `Set`, and `Error` are bridged to Foundation.

- For example, Swift strings are bridged to `NSString` and `NSMutableString`, as shown below:

```
let foundationStr = "Hello World!" as NSString
foundationStr.length                // 12
foundationStr.substring(from: 6)    // "World!"
foundationStr.substring(to: 5)      // "Hello"

let range = foundationStr.range(of: "World")  // (6, 5)
foundationStr.substring(with: range)          // "World"

let fruitText = "Apple, Pear, Banana" as NSString
let fruits2 = fruitText.components(separatedBy: ", ")
// ["Apple", "Pear", "Banana"]
```

- Foundation objects have powerful features; e.g., some can serialize/deserialize themselves to/from a file:

```
let s1 = "Apple, Pear, Banana"
let name = "/tmp/fruit.txt"

do {
    // Write s1 to a file
    try s1.write(toFile: name, atomically: true, encoding: .utf8)

    // Initialize a string with the file's contents
    let s2 = try String(contentsOfFile: name, encoding: .utf8)

    // Do something clever with s2...
}
catch _ {
    print("Unable to write to file \(name)”)
}
```

Classes

- Similar to structs, but support inheritance, and are reference types rather than value types
  - Typically allocated in heap
- Can contain properties, methods, and initializers/deinitializers (constructors/destructors)

```
class Animal {
    var isPet = false
}
// Declares Animal class

// Declares Dog to be subclass of Animal
class Dog: Animal {
    var name = ""

    func bark() {
        print("Woof, woof!")
    }

    func description() -> String {
        return "name: \(name), is pet: " + (isPet ? "yes" : "no")
    }
}
```

```
var rover = Dog()     // allocates a Dog instance
rover.name = "Rover"  // modifies name
rover.isPet = true;   // modifies isPet

print(rover.description())
// prints "name: Rover, is pet: yes"
```


Type Properties and Methods

- Swift types can have *type* properties and methods in addition to *instance* properties and methods.
  - Declare with **static** keyword
  - Class type methods and properties are inherited if declared with the **class** keyword (instead of **static**)

```
class Person
{
    static var eyeColors = [“blue”, “brown”, “hazel”]
    class var defaultAge = 21
    // ...

    static func printEyeColors() {
        print(eyeColors)
    }

    class func fetchPeople(path: String) -> [Person] {
        // ...
        // return some fetched people
    }
```

Access Control

- Each Xcode target defines a *module* — a single unit of code distribution that can be imported by another unit.

- Module names provide a namespace. By default, the module name is the same as the target name.

  - For example, if a class named **MyTests** is compiled in a target named **UnitTests**, the class's fully qualified name is **UnitTests.MyTests**.

- Access levels are relative to an entity's file and module.

Access Level	Visibility
open, public	everywhere
internal	all files in module *(default)*
fileprivate	file in which defined
private	entity in which defined

- Open access applies only to classes and their members.

Access/Entity	Subclassing
public classes	can be subclassed in module
public members	can be overridden in module
open classes	can be subclassed in module and wherever imported
open members	can be overridden in module and wherever imported

Lab 5: Classes and Protocols

1. Declare a class, **Person** that has two properties of type String named **firstName** and **lastName** with no default values.

    1.1. Write an initializer that takes a first and last name as arguments.

    1.2. Implement **debugDescription** to return the Person instance's first and last names and number of likes.

    1.3. Write a unit test to verify that you can instantiate and print a Person.

2. Declare a protocol named, **Likeable**, with a read-write property named **numberOfLikes** of type Int, and functions named **like** and **unlike** that take no arguments and have no return value. Then modify Person to adopt the **Likeable** protocol as follows:

    2.1. Add a **numberOfLikes** property with a default value of **0**.

    2.2. Add an extension that implements the **like** and **unlike** methods to increment and decrement **numberOfLikes**, but ensure that it never falls below zero.

    2.3. Modify **debugDescription** to include the number of likes in the string it returns.

    2.4. Write a unit test to verify that the new Likeable behavior works correctly.

3. Declare a protocol named **Friendable**, with read-only properties **friendID** of type Int, and **friends** of type array of **Friendable**, and functions **friend** and **unfriend**, both of which take a single argument of type **Friendable** and have no return value, Then modify Person to adopt the **Friendable** protocol as follows:

    3.1. Add a **friendID** property with a default value of **0**.

    3.2. Add a **friends** property that has an empty array as its default value.

    3.3. Add an extension that implements the **friend** function to add the provided friend to the **friends** array, and the **unfriend** function to remove the provided friend from the array.

    3.4. Write a unit test to verify that a Person can successfully friend and unfriend other Friendables.

CHAPTER THREE

Closures

Closures

- Swift functions can be passed as arguments and can be used as return values.
  - General form of closure syntax:

```
{ (parameters) -> return_type in
    // statements
}
```

- Example — passing a named function:

```
let fruit = ["Pear", "Apple", "Peach", "Banana"]

func ascending(s1: String, s2: String) -> Bool {
    return s1 < s2
}

let sortedFruit = fruit.sorted (ascending)
// value is ["Apple", "Banana", "Peach", "Pear"]
```

- Examples of syntactic flexibility:

```
// passing anonymous closure
let sortedFruit2 = fruit.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 < s2
})

// anonymous closure with streamlined syntax
let sortedFruit3 = fruit.sorted(by: { s1, s2 -> Bool in
    return s1 < s2
})

// trailing closure syntax
let sortedFruit4 = fruit.sorted { s1, s2 -> Bool in
    s1 < s2
}

// trailing closure with positional parameters
var sortedFruit5 = fruit.sorted { $0 < $1 }


// passing '<' function
var sortedFruit6 = fruit.sorted(by: <)
```


Closures as Arguments

- Closure arguments should always be declared at the end of the parameter list.

```
struct ContactInfo
{
    let phones = [
        "home": "202-123-4567",
        "work": "516-456-7890",
        "mobile": "914-789-1234",
        "other:": "914-456-7890"
    ]

    // Method that takes a closure as its only argument.
    // Returns a dictionary.
    func phones(matching: (String, String) -> Bool) -> [String: String]
    {
        var matchedPhones = [String: String]()
        for (key, value) in phones {
            if matching(key, value) {  // executes closure
                matchedPhones[key] = value
            }
        }
        return matchedPhones
    }
}


// Calling the phones(matching:) method with a trailing closure
//
let phonesMatchingAreaCodes = contact.phones() { key, value in
    value.hasPrefix("914")
}

print(phonesMatchingAreaCodes)
// produces ["other:": "914-456-7890", "mobile": "914-789-1234"]


// When the only parameter is a trailing closure, the parameter list
// can be omitted entirely
//
let daytimePhones = contact.phones { key, value in
    key == "work" || key == "mobile"
}

print(daytimePhones)
// produces ["mobile": "914-789-1234", "work": "516-456-7890"]
```


Closures Capture State

- Closures can automatically capture values from their enclosing scope.
  - Captured values are stored opaquely inside the closure.

```
let areaCode = "914"

let phonesMatchingCapturedValue = contact.phones { key, value in
    value.hasPrefix(areaCode)
}

print(phonesMatchingCapturedValue)
// produces ["other:": "914-456-7890", "mobile": "914-789-1234"]
```

- References can be captured, including references to **self**.
  - `self` keyword required when referencing **self** in closure.

```
// assume daytimeKeys is a property of self
var daytimeKeys = Set(["work", "mobile"])

// ...

let daytimePhonesWithCapturedValue = contact.phones { key, value in
    self.daytimeKeys.contains(key)
}

print(daytimePhonesWithCapturedValue)
// produces ["mobile": "914-789-1234", "work": "516-456-7890"]
```

- Optional *capture list* allows you to specify lifetime qualifiers.

```
let daytimePhonesWithCapturedValue = contact.phones {
    // unowned and weak qualifiers can help prevent retain cycles
    [unowned self] key, value in
    self.daytimeKeys.contains(key)
}

print(daytimePhonesWithCapturedValue)
// produces ["mobile": "914-789-1234", "work": "516-456-7890"]
```


map

- The **map** method operates on collections by applying a closure to each element successively.
  - Returns an array of the closure's return values.

```
let fruits = ["apple", "pear", "banana"]
let capitalizedFruits = fruits.map { $0.capitalizedString }
print(capitalizedFruits)
// prints "[Apple, Pear, Banana]"
```

- Powerful way to work with arrays of complex objects:

```
struct Grocery {
    let name: String
    let price: Double
    let quantity: Int
}

let groceries = [
    Grocery(name: "Apples", price: 0.65, quantity: 12),
    Grocery(name: "Milk", price: 1.25, quantity: 2),
    Grocery(name: "Crackers", price: 2.35, quantity: 3),
]

let costs = groceries.map {
    // a tuple containing current item's name and cost
    ($0.name, $0.price * Double($0.quantity))
}

print(costs)
// prints "[(Apples, 7.8), (Milk, 2.5), (Crackers, 7.05)]"
```

reduce

- The **reduce** method operates similarly to **map**, but instead of returning an array, returns a single value.
  - First argument is initial value
  - Second argument is a closure that returns the value of the current item in the collection combined with the current sum.

```
let ints = [1, 2, 3, 4, 5]

let sum = ints.reduce(0) {
    currSum, currVal in       // parameter list
    return currSum + currVal
}
// produces 15


// same as the above example, but with streamlined syntax:
let sum2 = ints.reduce(0) { $0 + $1 }
// ditto
let sum3 = ints.reduce(0, +)


// additional examples:

let factorial = ints.reduce(1, *)
// produces 120

let fruitsText = fruit.reduce("favorites: ") { "\($0)\($1), " }
// produces "favorites: apple, pear, banana, "
```

reduce (Continued)

- As with **map**, provides a powerful way to work with arrays of complex objects:

```
let total = groceries.reduce(0.0) { sum, item in
    sum + (item.price * Double(item.quantity))
}
// produces 17.35
```

- Applying **reduce** to array of tuples from an earlier **map** example:

```
let string = costs.reduce("Costs") { text, item in
    text + "name: \(item.0), cost: \(item.1)\n"
}

print(string)
// name: Apples, cost: 7.8
// name: Milk, cost: 2.5
// name: Crackers, cost: 7.05


// fancier version of preceding example:

let text = costs.reduce("Costs\n==============\n") {
    $0 + String(format: "%8s%6.2f\n",
         NSString(string: $1.0).UTF8String, $1.1)
}

print(text)
Costs
==============
  Apples  7.80
    Milk  2.50
Crackers  7.05
```


filter

- The **filter** method returns an array of objects that match the condition specified by its closure argument.

```
let people = [
    Person(("Fred", "Smith", 27)),
    Person(("Janet", "Wade", 31)),
    Person(("Gale", "Dee", 42)),
    Person(("Jan", "Grey", 29)),
]

let under30 = people.filter { $0.age < 30 }
// produces an array containing Fred Smith and Jan Grey
```

CHAPTER FOUR

Optionals and Error Handling

The Optional Enum

- Null is not a legal value in Swift.
- Use optional instances to wrap any values that can potentially be null.
  - `none` case represents absence of a value
  - `some` case wraps a non-null associated value

Declaration of Optional enum from Swift Standard Library

```
public enum Optional<Wrapped> : ExpressibleByNilLiteral {

    /// The absence of a value.
    ///
    /// In code, the absence of a value is typically written using the `nil`
    /// literal rather than the explicit `.none` enumeration case.
    case none

    /// The presence of a value, stored as `Wrapped`.
    case some(Wrapped)

    /// Creates an instance that stores the given value.
    public init(_ some: Wrapped)

    ...
}
```

Optional Enum Syntax

- Using enum syntax to define optionals

```
let wrappedText: Optional<String> = Optional.some("lol")

// Initializer for Int returns an optional
let wrappedX: Optional<Int> = Int("12")
```

- Using enum syntax to unwrap optionals

```
// Using enumeration case pattern to unwrap an optional
if case .some(let text) = wrappedText {
    print(text.uppercased())
}
// Prints "LOL"

// Alternate syntax for enumeration case pattern
if case let .some(x) = wrappedX {
    print("x squared is \(x * x)")
}
// Prints "x squared is 144"
```

Optional Pattern Syntax

- Using optional pattern to define optionals

```
// String? is shorthand for Optional<String>
let wrappedText: String? = "lol"

// Int? is inferred type of wrappedX
let wrappedX = Int("12")
```

- Using optional pattern to unwrap optionals (*optional binding*)

```
// Using optional pattern to unwrap an optional
if case let text? = wrappedText {
    print(text.uppercased())
}

// case let ? can be inferred by compiler

if let text = wrappedText {
    print(text.uppercased())
}

if let x = wrappedX {
    print("x squared is \(x * x)")
}
```

Optional Pattern With for Loop

- Working with an array of optionals

```
// Defining an array of optionals
let wrappedWords: [String?] = [nil, "Apple", nil, "Orange"]

// Print non-nil values
for word in wrappedWords {
    if let unwrappedWord = word {
        print(unwrappedWord)
    }
}
```

- Simplifying looping code

```
// Test expression conditionally unwraps optional
for case .some(let word) in wrappedWords {
    print(word)
}

// Streamlined test expression
for case let word? in wrappedWords {
    print(word)
}
```

Optional Binding With guard

- **guard–let** is an alternative to **if–let**

```
func squared(numericString: String) -> String?
{
    guard let value = Double(numericString) else { return nil }
    let square = value * value
    return square.description
}


if let s = squared(numericString: "12") {
    print(s)
}
// prints "144.0"
```

Forced Unwrapping

- Use the exclamation point operator to force unwrap an optional.

- Caution: inherently unsafe.

```
if name != nil {
    print("name is \(name)")
    // prints "name is Optional("Fred")"

    print("name is " + name!)  // forced unwrap (unsafe)
    // prints "name is Fred"
}
```

Guard vs. the Pyramid of Doom

- Consider the 'pyramid of doom' style of nested **if–let** statements in the following example:

```
func format1(person: Person?) -> String
{
    if let p = person {
        if let name = p.fullName {
            if let ageStr = p.age, let age = Int(ageStr) {
                return "\(name), age: \(age)"
            } else { return name }
        } else { return "missing name" }
    }
    return "person cannot be nil"
}
```

- The previous example, rewritten using **guard–let**:

```
func format2(person: Person?) -> String
{
    guard let p = person else { return "person cannot be nil" }
    guard let name = p.fullName else { return "missing name" }
    guard let ageStr = p.age, let age = Int(ageStr) else {
        return name
    }
    return "\(name), age: \(age)"
}
```

Casts

- Downcast with `as`, `as?`, or `as!`

Downcast with forced unwrap

```
var words: [Any] = ["Hello", "World"]
let word = words[0] as! String   // forced downcast
print(word)
// prints "Hello"

words[0] = 1                     // hm...
print(words[0] as! String)       // doh!
```

Optional downcast with safe unwrap using optional pattern

```
if let thing = words[0] as? Int {    // optional downcast
    print(thing)
}
// prints "1"
```

- Pattern matching with downcasts in a switch statement:

```
for currVal in words {
    switch (currVal) {
    case let value as Int:    print("\(value / 6)")
    case let value as String: print("Hi \(value)!")
    case let value as Double: print("\(value * 2)")
    default:                  print("value is \(currVal)")
    }
}
// "7"
// "Hi Fred!"
```

Type Checking

- Check type with `is`, `is?`, or `is!`

```
var objects: [Any] = [42, "Fred", 3.5]
for object in objects {
    if object is Int    { print("The answer is \(object)") }
    if object is String { print("Hi \(object), how are you?") }
}
// "The answer is 42"
// "Hi Fred, how are you?"
```


Lab 6: Optionals

1. Declare a struct, **Address**, with three readonly stored properties of type String, **street, city,** and **state**, and a readwrite property of type Optional<String>, **street2**, initialized to **nil**. Write a custom initializer that takes all four parameters and provides a default value of **nil** for **street2**. Add a **fullStreet** computed property that returns **street** if **street2** is **nil**, otherwise a String containing **street** followed by a comma, and then **street2**. Write a unit test to test the behavior of **fullStreet**.

2. Declare a class, **Customer** that has stored properties, **name**, of type String, and **address**, of type Optional<Address>. Write any necessary initializers, and make both **Customer** and **Address** conform to **CustomStringConvertible**, and provide appropriate implementations of **description**. Write a unit test to test initializing and describing a Customer.

3. Add an extension to Array, constrained to elements of the Customer type. In the extension, define a method **customer(named:)** that takes one argument of type String. The method should enumerate an array of Customer objects, returning the first one whose name matches the provided name. Write a unit test to the method works as expected on an array of customers.

Implicitly Unwrapped Optionals

- Exclamation point can be used as qualifier for typing an optional as *implicitly unwrapped*.
  - The resulting optional can then be used as if it were unwrapped.
  - The developer is responsible for ensuring that the value will not be directly accessed when **nil** (other than testing for nil).

```
// Given lastName is an implicitly unwrapped optional...
var lastName: String! = "Smith"

// The following line prints "Fred Smith"
print("Fred " + lastName)

// However, if lastName is set to nil...
lastName = nil

// The identical line throws a fatal exception
print("Fred " + lastName)
```

The Nil Coalescing Operator

- The *nil coalescing operator is* **??**.
- Combines ternary expression and optional unwrapping.

```
let special: Double?
// ...

let discount = special ?? 0.15
// Assigns 0.15 to discount if special is nil,
// otherwise assigns unwrapped value of special

// The preceding expression is shorthand for:
let discount = special == nil ? 0.15 : special!
```

- Can be used as part of a larger expression:

```
let Unknown = "Unknown"
let firstName: String? = "Fred"
let lastName: String? = "nil"

var description: String {
    return "first: \(firstName ?? Unknown), "
        +  "last: \(lastName ?? Unknown)"
}

print(description)

// prints "first: Fred, last: Unknown"
```

Optional Chaining

- Combines optional unwrapping and property access.

Accessing values by chaining through optional properties

```
var fred: Person? = Person(firstName: "Fred", lastName: "Smith")
// Defines an optional that wraps an instance of Person

print(fred?.firstName)
// Prints "Optional("Fred")

print(fred?.dog?.toys)
// Prints "Optional(["Ball, "Frisbee"])
```

Mutating values by chaining through optional properties

```
fred?.dog = Dog(name: "Fido", toys: ["Ball", "Frisbee"])
// Modifies dog property of optional Person

fred?.dog?.toys = ["Kong", "Rope"]
// Modifies toys property of optional dog property
// of optional Person
```


Error Handling

- Swift throws errors rather than exceptions.
- Errors defined by enums conforming to the `Error` protocol.

```
enum MathError: Error {
    case zeroDivide
    case overflow
}
```

- Thrown errors don't unwind the stack; instead they trigger an early return that incorporates an error object.
- Methods and functions that throw must be annotated with the **throws** keyword

```
extension Double
{
    func divided(by denominator: Double) throws -> Double {
        if denominator == 0.0 {
            throw MathError.zeroDivide
        }
        return self / denominator
    }
}
```


Catching Thrown Errors

- Use a **do–catch** block and the **try** keyword to perform explicit error handling.

```
do {
    let result = try 42.divided(by: 0)
    print("result is \(result)")
}
catch MathError.zeroDivide {
    print("Zero divide")
}
// prints "Zero divide"
```

- A **do–catch** block can include multiple **try** and **catch** expressions.

```
do {
    let result1 = try 42.divided(by: 2)
    print("result 1 is \(result1)")

    let result2 = try result1.divided(by: 3)
    print("result 2 is \(result2)")
}
catch MathError.zeroDivide {
    print("Zero divide")
}
catch MathError.overflow {
    print("Overflow")
}
catch {
    print ("Unexpected error")
}
// result 1 is 21.0
// result 2 is 7.0
```


Optional Variants of try

- Use the optional variants **try?** or **try!** instead of a **do-catch** block when you don't need custom error handling.

Implicitly unwrapped `try`

```
let x = try! 12.divided(by: 1.25)
print(x)
// Prints "9.6"

let xxx = try! 12.divided(by: 0)
// Throws a fatal error
```

Optional `try`

```
guard let y = try? 12.divided(by: 1.5) else { return }
print(y)
// Prints "8.0"

guard let z = try? 12.divided(by: 0) else { return }
// Does nothing
```

Cleaning Up With `defer`

- Use **defer** blocks to specify cleanup actions.

```
enum FileError: Error {
    case unknown
    case nonexistent(String) // note: case declared with associated value
}

func showFile(atPath path: String) throws {
    // initialize file handle
    guard let fileHandle = FileHandle(forReadingAtPath: path) else {
         // note: throw error with associated value
        throw FileError.nonexistent("No file at path \(path)")
    }

    // clean up file handle at end of current scope
    defer {
        fileHandle.closeFile()
    }

    // use file handle...
    let data = fileHandle.readDataToEndOfFile()
    FileHandle.standardOutput.write(data)
}

func foo()
{
    do {
        try showFile(atPath: "/tmp/README.md")
    }
    catch FileError.nonexistent(let message) {
        print(message) // note: uses associated value
    }
    catch is FileError {
        print("Some other file error occurred.")
    }
    catch {
        print("Unexpected error.")
    }
}
```

- Deferred blocks are pushed on a stack for the current scope.
  - Items are popped off the stack and executed when the current scope exits.

CHAPTER FIVE

Bridging to Objective-C

Bridging Within a Target

- Create a *bridging header* to make Objective-C headers visible to Swift code in the same target.
  - Add file named ***module_name*-Bridging-Header.h**
  - Import desired ObjC headers

```
// Headers to expose to Swift
//
#import "Person.h"
#import "PersonViewController.h"
```

- Xcode generates a single Objective-C header for all Swift source code compiled in a given module.
  - Generated file name is ***module_name*-Swift.h**
  - Import the generated header in any Objective-C file that references items in the Swift source.

```
// Import our own module's Swift headers
//
#import "MyApp-Swift.h"
```


Bridging to Swift Frameworks

- Add module import statements to any Objective-C source files that refer to Swift framework APIs.

```
// Module import for Swift framework target
//
@import SwiftComponents;
```

Data Types

- `id` data type is translated as `Any`, `Class` as `AnyClass`.
- Lightweight generic support added to Objective-C to allow improved Swift translation.

```
// Bridged as `var toys: [Toy]`
//
// Note: non-parameterized arrays are bridged as `[Any]`
//
@property (strong, nonatomic) NSArray<Toy *> *toys;
```

- Many commonly-used Foundation types such as **NSString**, **NSArray**, and **NSData**, are bridged to Swift value types (in this case **String**, **Array**, and **Data**).
- Many Cocoa library structures are also bridged as Swift value types.

```
// Initialize a CGRect
var rect1 = CGRect(x: 0, y: 0, width: 80, height: 30)

// Mutate
rect1.insetInPlace(dx: 5, dy: 8)
```

Nullability Annotations

- Objective-C declarations can be annotated with nullable, non null, and null_resettable to specify translation to Swift optional vs. non-optional types:

```
@interface Pet : NSObject

// Bridged as `init(name: String?, type: PetType)`
- (nonnull instancetype)initWithName:(nullable NSString *)name;

// Bridged as `var name: String`
@property (nonnull, strong, nonatomic) NSString *name;
```

- Use `NS_ASSUME_NONNULL…` macros to annotate regions of an Objective-C header.

```
NS_ASSUME_NONNULL_BEGIN

@property (nonatomic) PetType type;
@property (nonnull, strong, nonatomic) NSString *name;

@property (nonnull, strong, nonatomic) NSArray<Toy *> *toys;

NS_ASSUME_NONNULL_END
```