CS 329E
Elements of Mobile Computing

Spring 2018
University of Texas at Austin

Lecture 2

Agenda

- Development components
- Swift
- Creating an Xcode Command Line app
- Creating a Zip file on a Mac
- Homework 1

Development Components

Development Components

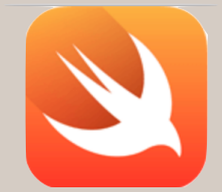
What you need to develop iOS applications:

- Computer - Mac

- IDE - Xcode

  - Editor, compiler, linker, debugger, UI designer

- Language - Swift



- Supporting frameworks - Foundation, UIKit, etc
  - Come with Xcode

Swift

Swift

Topics:

- What is Swift?
- Data Types
- Variables
- Flow Control
- Functions
- Classes

Swift
What is Swift?

Swift

What is Swift?

A new programming language for developing macOS, iOS, watchOS and tvOS applications.

The intent was to take the best of C and Objective-C and not worry about C compatability.

A language that makes programming easier (productivity), more flexible (expandability) and more fun.

Swift

- Provides seamless access to Cocoa frameworks
  - The frameworks were modernized and standardized to properly support Swift; which allowed for adding new features to Objective-C, e.g. blocks
- Provides mix-and-match interoperability with Objective-C code
- Apple considers Swift a *systems* programming language
  - Meaning, there's power in the language to do system-level stuff, e.g. operating systems
- ***Everything is an object!***
- ***Swift is open source!***

Swift

A complete Swift program:

print(“Hello World!”) <== *no semicolon!!*

Output: Hello World!

Ok. Fine. But this is way too simple for a real-world program, where you have lots of classes, etc.

I want to know where a program starts (like ‘main’ for Objective-C) and how it gets into the rest of your application code.

Swift

So, where is the *entry point* to the program?
(Objective-C had the main function)

The entry point for a Swift program is the code at *global scope* - code that is outside any function.

- For a Command Line application the only file allowed to have executable code at global-scope is **main.swift**
- For iOS applications no files are allowed to have executable *user* code at global-scope; the entry point is in **AppDelegate.swift**

Swift

So:

When you create a Command Line application project, Xcode creates a **main.swift** file and adds it to the project.

When you create an iOS Application project, Xcode creates an **AppDelegate.swift** file (among others) and adds it to the project.

Swift
Data Types

Swift

Data Types:

- Data types define what *kind of thing* a variable is
- You can use *data type annotation* when declaring a variable:
    - Data type annotation is a colon followed by the data type
        - Example: var name:String = “Joe”
- Data types are not “required” when defining variables; the type is *inferred* by the value you assign to the variable
    - Example: var name = “Joe” // name becomes a String

Swift

Data Types:

- Data types come *<u>after</u>* the variable name - which is opposite of most languages

var name**:String** = “Joe”

But this is necessary because they are <u>optional</u>.

Swift

Data Types:

- Basic data types:
  - Int var total:Int = 100
  - Int, Int8, Int16, Int32, Int64
  - UInt, UInt8, UInt16, UInt32, UInt64
- Float (32 bits) var salesMonth:Float = 350.00
- Double (64 bits) var salesYear:Double = 2500.00
- String var name:String = “Joe”
- Bool var isDone:Bool = false

Swift

Data Types:

- Bool data type:
  - Must be ‘true’ or ‘false’

```
var isDone:Bool = false
isDone = true

// error: Cannot assign a value of type ‘Int’ to a value of type ‘Bool’
isDone = 55
```

Although, internally false is a value of zero and true is everything else.

Swift

Data Types:

- In Swift, there are two *kinds* (groupings) of data types:
  - Named types
  - Compound types

Swift

Data Types:

- Named types:
  - A type that can be given a particular name when it is defined
  - Includes classes, structures, enumerations and protocols
  - Examples: Int, myClass (user-defined type)

Swift

Data Types:

- Named types:
  - Data types that are normally considered basic or primitive in other languages - such as types that represent numbers, characters, and strings - are actually named types, defined and implemented in the Swift standard library using structures.
  - Because they are named types, you can extend their behavior.

Swift

Data Types:

- Compound types:
  - A type without a name
  - There are two compound types:
    - Function types
    - Tuple types
  - May contain named types and other compound types

Swift

Data Types:

- Compound type - Function types:
  - Represents the type of a function, method or closure
  - Consists of a parameter type and return type separated by an arrow (->)

parameter-type -> return-type

(essentially what you pass in and what is returned)

Swift

Data Types:

- Compound type - Function types:
  - Because the parameter type and return type can be a tuple type, function types support functions and methods that *take multiple parameters* and ***return multiple values***
    - Passing N values into a function has been around for a long time
    - Returning N values as the value of a function is a HUGE new capability!

Swift

Data Types:

- Compound type - Function types - examples:
  - Function that takes an integer and returns a string

    Int -> String

  - A function that takes a float and integer and returns an integer and string

    (Float, Int) -> (Int, String)

Swift

Data Types:

- Compound type - Tuple types:
  - A tuple is a comma-separated list of zero or more types, enclosed in parenthesis
  - Unnamed

```
let origin:(Int, Int) = (10, 20)
let stuff:(Int, (Int, Int)) = (10, (30, 40))
```

Swift

Data Types:

- type alias
  - Defines an alternative name for an existing type
  - Useful when you want to refer to an existing type by a name that is contextually more appropriate

// Define an alias type for a tuple with two integers
typealias Point = (Int, Int)

var origin:Point = (10, 20) <— Point is more understandable

var origin:(Int, Int) = (10, 20) <— instead of this

Swift
Variables

Swift

Variables:

- Used throughout a program to store values
- A name that represents a memory location
- Use var for a mutable variable

  var name = “Joe”
- Use let for an immutable variable; a constant

  let name = “Joe”

Note: There is no data type defined in these examples!

Swift

Variables:

Once you set a variable to a value, the data type is set and can’t be changed.

```
var name = “Joe”

// error: Cannot assign a value of type 'Int' to
// a value of type 'String'
name = 55
```

Swift

Variables:

Of course, you can always specify a variable's data type. This is called *annotating* a variable.

```
var name:String = "Joe"
```

Swift

Variables - constants:

A constant can only be set once.

```
let name = “Joe”

// error: Cannot assign to 'let' value 'name'
name = “Sam”
```

Swift

Variables - constants:

A constant does NOT need to be initialized when it is declared, unlike most other languages. But, if you do not set a value, you must define the data type.

```
// error: Type annotation missing in pattern
let name

name = “Sam”
```

Swift

Variables - constants:

```
// “Type annotation missing” error fixed by
// adding data type
let name:String

name = “Sam”
```

Swift

Variables - tuples:

- A tuple is a type with zero or more elements enclosed in parenthesis
- You can name each element in the tuple
- You (currently) can not iterate over them, like arrays
- The number of elements in a tuple cannot change, once it is set

Swift

Variables - tuples:

- A tuple can have any combination of data types

```
// The data type here is: (String, Int, Float)
// Three total entries.
let myStuff = ("A", 3, 4.5)

// The data type here is: (String, (Int, Float))
// Two total entries in the (outermost) tuple.
// That is, a string, followed by a tuple.
let myStuff = ("A", (3, 4.5))
```

Swift

Variables - tuples:

- You can access the elements in a tuple using simple indexing using dot notation

```
let origin:(Int, Int) = (10, 20)
print(origin)       // output: (10, 20)
print(origin.0)     // output: 10
print(origin.1)     // output: 20
```

Swift

Variables - tuples:

- You cannot use array-like indexing

```
let origin:(Int, Int) = (10, 20)

// error: (Int, Int) does not have a member
// named ‘subscript’
print(origin[0])
```

Swift

Variables - tuples:

- You can modify any given element

```
var origin:(Int, Int) = (10, 20)
print(origin)       // output: (10, 20)
print(origin.0)     // output: 10

origin.0 = 100
print(origin.0)     // output: 100
```

Swift

Variables - tuples:

- You can decompose (break apart) the tuple to assign each element to a separate variable

```
var origin:(Int, Int) = (10, 20)
var (tot1, tot2) = origin
print(tot1)         // output: 10
print(tot2)         // output: 20
```

Swift

Variables - tuples:

- You cannot add any elements once the tuple is set

```
var origin:(Int, Int) = (10, 20)
// error: (Int, Int) does not have a member named '2'
origin.2 = 50
```

Swift

Variables - tuples:

- You can name any number of elements in a tuple
  - In this way, it makes it like a cheap struct

```
var myStuff = (prefix:”A”, tot:3, avg:4.5)
print(myStuff.prefix)   // output: A
myStuff.avg = 5.3

var myStuff = (”A”, 3, avg:4.5)  // only one named
```

Swift

Variables - tuples:

- Two special cases:
  - A tuple with zero elements
  - A tuple with one element

Swift

Variables - tuples - zero elements:

```
// Warning: Variable 'myStuff' inferred to have
// type '()', which may be unexpected
var myStuff = ()
print(myStuff)   // output: ()
```

It doesn’t make much sense to create an empty tuple, since you can’t add any elements once it’s set.

Swift

Variables - tuples - one element:

- There is no one-element tuple (based on the documentation)

These are all the same - a variable of type Int. The parenthesis are superfluous.

```
var a: Int
var b: (Int)
var c: (((((((((Int)))))))))
var d = 42
var e = (42)
var f = (((((((((42)))))))))
```

Swift
Flow Control

Swift

Flow Control:

- operators
- if statement
- for loop
- while loop
- repeat-while loop

Swift

Flow Control - operators:

- standard set of logical and relational operators
- logical operators
  - && (and)
  - || (or)
  - ! (not)
- relational operators
  - > (greater than)
  - < (less than)
  - >= (greater than or equal to)
  - <= (less than or equal to)
  - != (not equal to)
  - == (equal to)

Swift

Flow Control - if statement:

- standard if statement structures

```
var v1 = 55
var v2 = 66

if v2 > v1 {    <== parenthesis not required, but ok
    print("v2 > v2")
} else {
    print("v1 > v2")
}
```

Swift

Flow Control - for loop:

- Two kinds:
  - Executes the loop body a fixed number of times
  - Executes the loop body until a condition is met

Swift

Flow Control - for loop:

- Execute the loop body a fixed number of times for an index range

```
for idx in 1...3 {
    print("idx is \(idx)”)   <— ‘\(idx)’ substitutes the value of idx
}
```

Output:
```
idx is 1
idx is 2
idx is 3
```

Swift

Flow Control - for loop:

- Execute the loop body a fixed number of times for a *collection*

```
let names = ["Anna", "Alex", "Brian"]
for name in names {
    print("Hello, \(name)!")
}
```

Output:
```
Hello, Anna!
Hello, Alex!
Hello, Brian!
```

Swift

Flow Control - for loop:

- Execute the loop body until a condition is met
  - Traditional C/C++ pre-test *for* loop

```
for var index = 0; index < 3; index += 1 {
    print("index is \(index)")
}
```

Output:
```
index is 0
index is 1
index is 2
```

Swift

Flow Control - while loop:

- Execute the loop body until a condition is met
  - Traditional C/C++ pre-test *while* loop

```
var index = 0
while index < 3 {
    print("index is \(index)")
    index += 1
}
```

Output:
```
index is 0
index is 1
index is 2
```

Swift

Flow Control - repeat-while loop:

- Execute the loop body until a condition is met
  - Traditional C/C++ post-test *do-while* loop

```
var index = 0
repeat {
    print("index is \(index)")
    index += 1
} while index < 3
```

Output:
```
index is 0
index is 1
index is 2
```

Swift Functions

Swift

Defining a function:

- In this context, we're talking about stand-alone functions, i.e. not within classes

```
func <function-name>([<argument-list>]) [-> <return-type>]
{
    [<code>]
}
```

Square brackets [] means optional:

- argument-list
- return-type
- code

Swift

Defining a function:

- Simplest function - one with no arguments

```
func displayMessage()
{
  print(“Hello again!”)
}
```

Swift

Defining a function:

- The argument list is composed of zero or more arguments, separated by commas
- Each argument consists of: a name, a colon, a type

```
func callMe(name: String)
{
  print(“Name: \(name)”)
}
```

Swift

Defining a function:

- Traditional function call syntax

```
func callMe(name: String)
{
  print(“Name: \(name)”)
}
```

Usage:

```
callMe(name: “Joe”)     // output: Name: Joe
```

Swift

Internal versus *external* function argument names:

- Each function argument can have 2 names!
- One internal and one external.
  - The internal argument name is used to reference the passed in value within the scope of the function.
  - The external argument name is used when calling the function.

Swift

Internal versus *external* function argument names:

- If you only define one it is both internal and external

```
func callMe(name: String)
{
  print(“Name: \(name)”)
}
```

So, here, ‘name’ is both the internal and external argument name.

Swift

Internal versus *external* function argument names:

- You can explicitly define an external argument name
  - Don't define explicit external argument names unless there is a good reason

```
func callMe(extName name: String)
{
  print("Name: \(name)")
}

Usage:
callMe(extName:"Joe")     // output: Name: Joe
```

Swift

Defining a function:

- You can pick and choose which arguments have explicit external names

```
func callMe1(extName name: String, age: Int) {
  print(“Name: \(name), Age:\(age)”)
}
func callMe2(name: String, extAge age: Int) {
  print(“Name: \(name), Age:\(age)”)
}

Usage:
callMe1(extName:”Joe”, age: 25)  // output: Name: Joe, Age: 25
callMe2(name: ”Joe”, extAge:25)  // output: Name: Joe, Age: 25
```

Swift

Defining a function:

- When calling a function, you can not reorder arguments, even though they all have names

```
func callMe(name: String, age: Int)
{
  print(“Name: \(name), Age:\(age)”)
}

Usage:
// error: Argument ‘name’ must precede argument ‘age’
callMe(age: 27, name:”Joe”)
```

Swift

Defining a function:

- Returning a value - the return type is defined at the end of the function header

```
// Returns a composed string
func sayHiBack(msg:String) -> String
{
    return ("Hi \(msg)")
}

Usage:
print(sayHiBack(“Joe”))   // output:  Hi Joe
```

Swift

Defining a function:

- Now we can take advantage of returning N items!!

```
// Returns a tuple consisting of an integer and a string
func sayHiBack(msg:String) -> (Int, String)
{
    return (5, ”Hi \(msg)")
}

Usage:
print(msg:sayHiBack(“Joe”))   // output:  (5, Hi Joe)
```

Swift

Defining a function:

```
// Because this function returns a tuple, you can name
// any of the elements of the tuple
func sayHiBack(msg:String) -> (count:Int, label:String)
{
    return (5, ”Hi \(msg)")
}

Usage:
result = sayHiBack(“Joe”)
println(result.label)   // output: Hi Joe
```

Swift

Defining a function:

- Decomposing function return values

```
func sayHiBack(msg:String) -> (count:Int, label:String)
{
    return (5, ”Hi \(msg)")
}
```

Usage:

```
// You can decompose the result of the function
(tot, displayMsg) = sayHiBack(“Joe”)
println(displayMsg)   // output: Hi Joe
```

Swift Classes

Swift

Defining a class:

Basic class definition:

```
class <name> {
    <properties>
    <methods>
}
```

Swift

Defining a class:

```
class Person {
    // Properties
    var firstName:String = "<NotSet>"
    var lastName:String = "<NotSet>"

    // Methods
    init(firstName:String, lastName:String) { < == no ‘func’ in front of ‘init’
        self.firstName = firstName
        self.lastName = lastName
    }

    func description() -> String {
        return "First Name: \(firstName), Last Name: \(lastName)"
    }
}
```

Swift

Creating an instance of a class:

- Each instance has it’s own memory and therefore it’s own set of properties

```
p1 = Person(firstName:”Sam”, lastName:”Smith”)

// Calling a public method of the class
p1.description()
```

Swift

Defining a class-level (known in Swift as a type-level) method - prefix with ‘class’ in method header:

```
class Person {
    // Properties
    var firstName:String = "<NotSet>"
    var lastName:String = "<NotSet>"

    // Methods
    init(firstName:String, lastName:String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    class func createOne(firstName:String, lastName:String) -> Person {
        return Person(firstName: firstName, lastName: lastName)
    }

    func description() -> String {
        return "First Name: \(firstName), Last Name: \(lastName)"
    }
}
```

Swift

Calling a class-level method:

```
var p1 = Person.createOne(firstName:”Sam”, lastName:”Smith”)
```

Swift

Defining a class-level property - prefix with **static**:

```
class Person {
    // Properties
    var firstName:String = "<NotSet>"
    var lastName:String = "<NotSet>"
    static var prefix:String = "UT:"

    // Methods
    init(firstName:String, lastName:String) {
        self.firstName = firstName
        self.lastName = lastName
    }

    class func createOne(firstName:String, lastName:String) -> Person {
        return Person(firstName: firstName, lastName: lastName)
    }

    func description() -> String {
        return "First Name: \(firstName), Last Name: \(lastName)"
    }
}
```

Swift

Accessing a class-level property:

class-name, dot, property-name

```
println(Person.prefix)   // output: UT:
```

Swift

Defining private instance *properties* in a class - prefix with **private**:

- Means no code outside of class code can access that property

```
class Person {
    // Properties
    var firstName:String = "<NotSet>"
    private var lastName:String = "<NotSet>"

   // Methods
    init(firstName:String, lastName:String) {
        self.firstName = firstName
        self.lastName = lastName
    }

   func description() -> String {
        return "First Name: \(firstName), Last Name: \(lastName)"
    }
}
```

Swift

Trying to access a private instance property from outside the class:

```
p1 = Person(firstName:”Sam”, lastName:”Smith”)

// Ok, because this property is public
p1.firstName = “Sally”

// error: ‘Person’ does not have a member named ‘lastName’
p1.lastName = “Jefferson”
```

Swift

Defining private instance *methods* in a class - prefix with **private**:

- Means no code outside of class code can access that method

```
class Person {
    // Properties
    var firstName:String = "<NotSet>"
    var lastName:String = "<NotSet>"

   // Methods
    init(firstName:String, lastName:String) {
        self.firstName = firstName
        self.lastName = lastName
    }

   private func description() -> String {
        return "First Name: \(firstName), Last Name: \(lastName)"
    }
}
```

Swift

Trying to access a private instance method from outside the class:

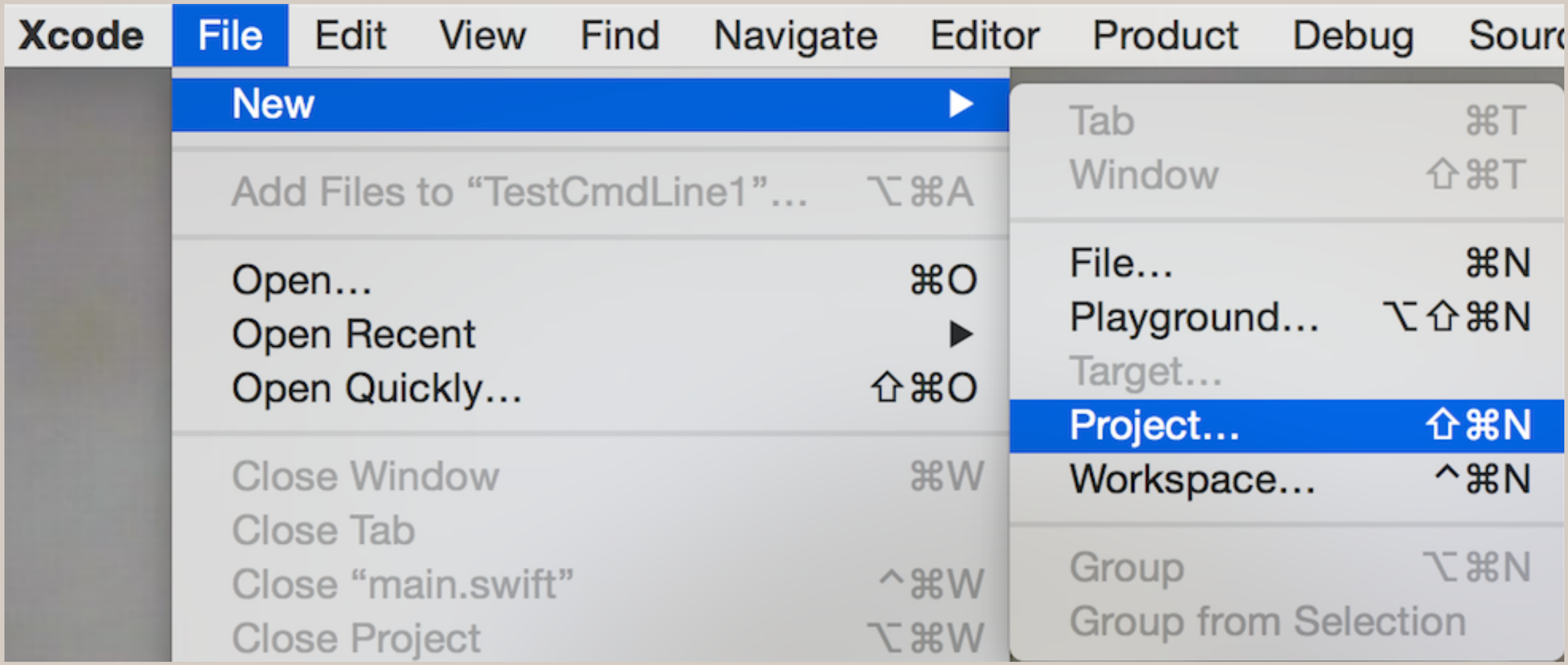
```
p1 = Person(firstName:”Sam”, lastName:”Smith”)

// error: ‘Person’ does not have a member named ‘description’
println(p1.description())
```

Creating an Xcode Command Line App

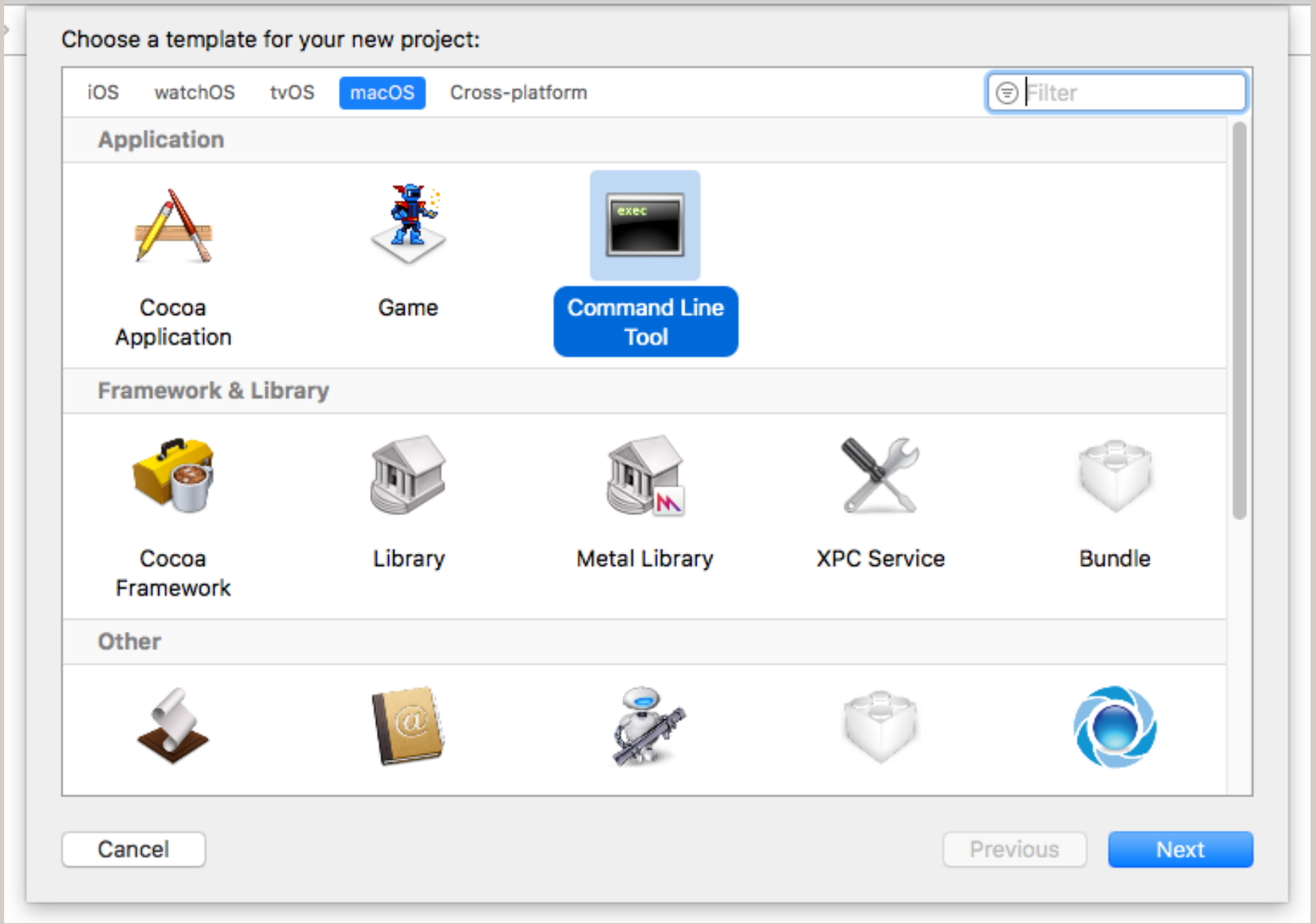
Creating a CL Application

Create a new project:



Creating a CL Application

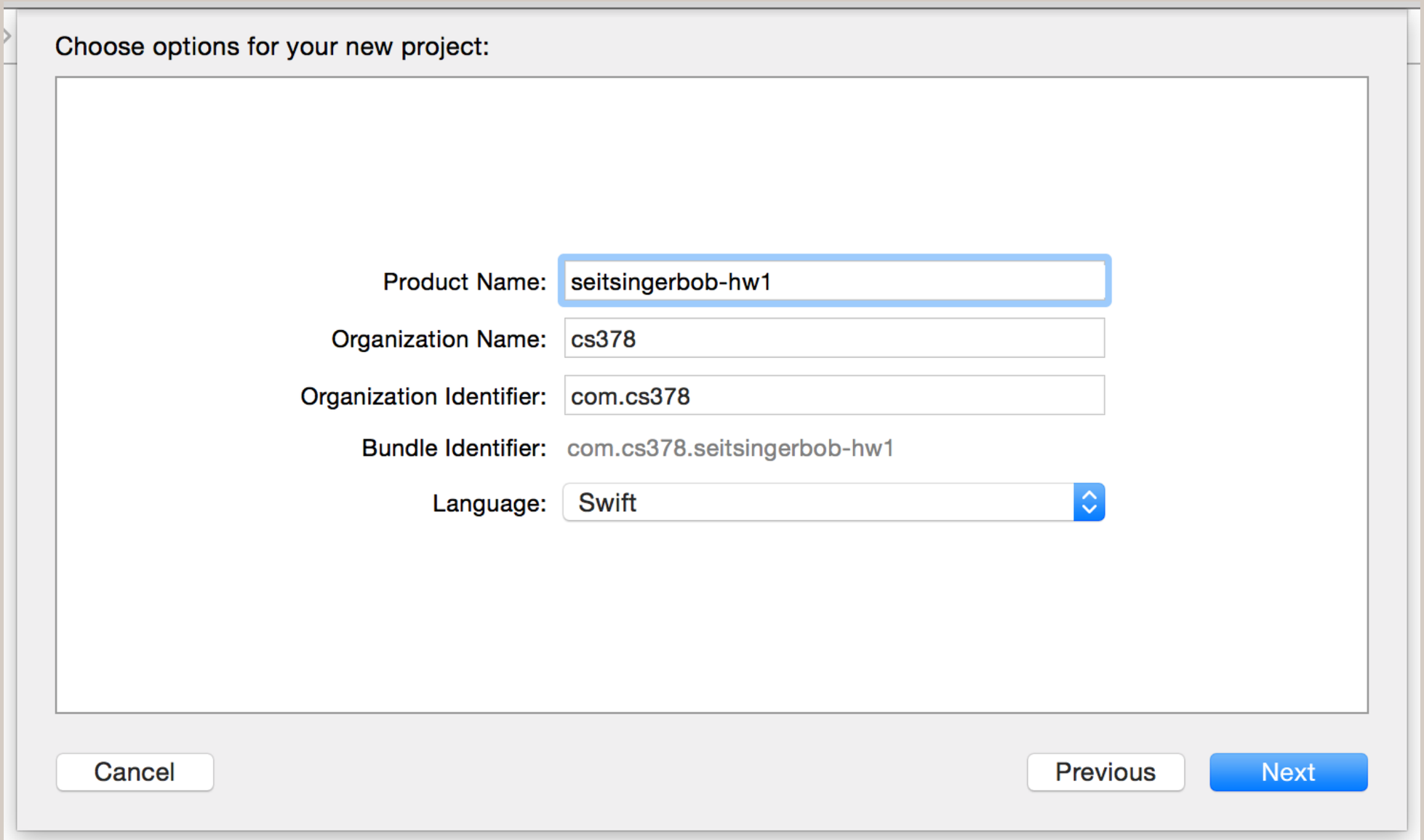
Select an OS X Command Line Tool project:



Creating a CL Application

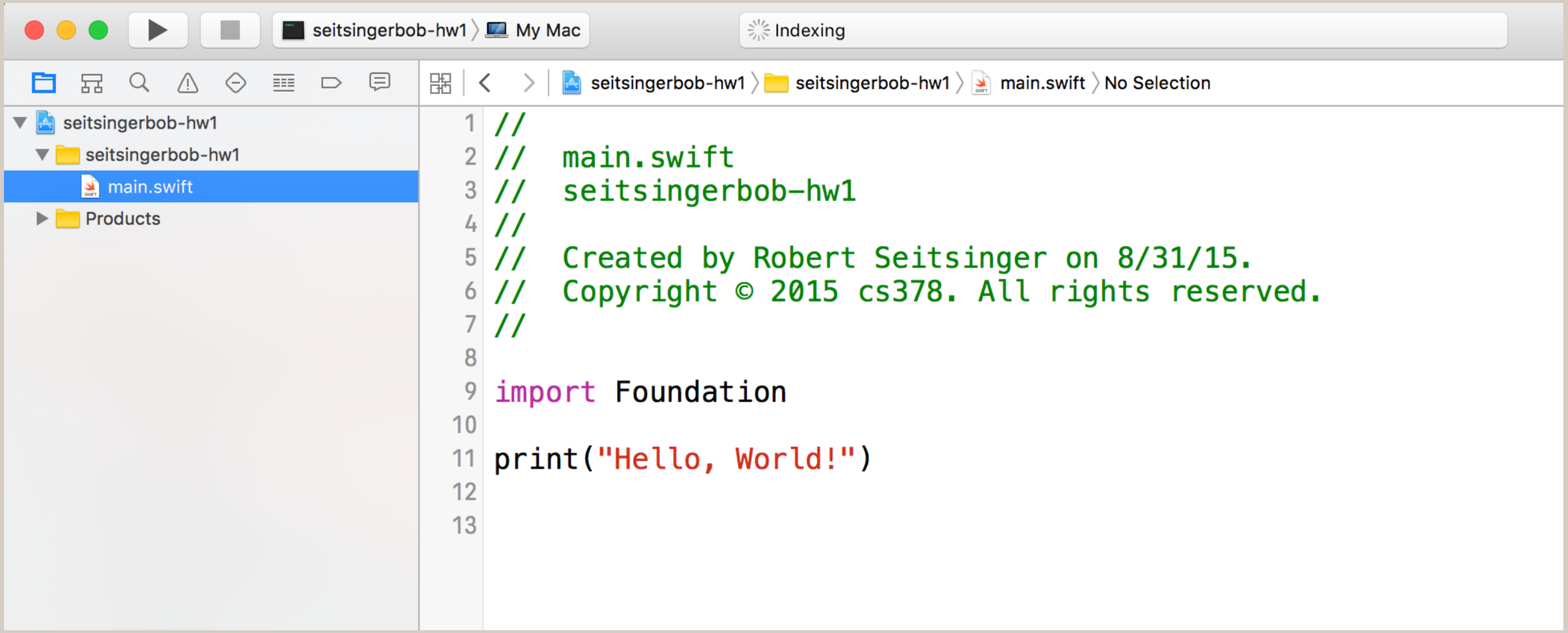
Enter project-specific options:

- Notice the Product Name - **<lastname><firstname>-hw<?>**



Creating a CL Application

main.swift - the automatically created source file



Creating a CL Application

Change main.swift to have this basic layout.

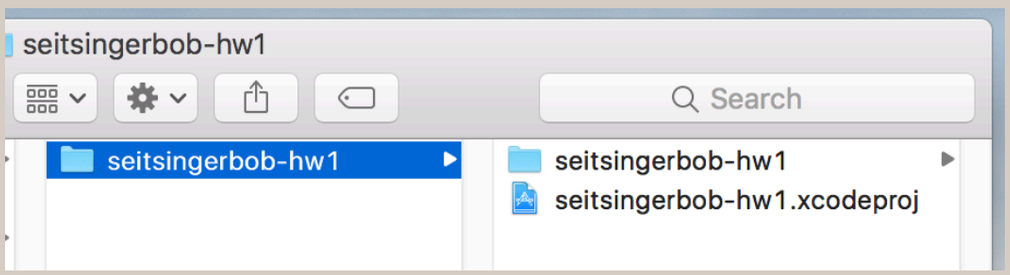
```
1  //
2  //  main.swift
3  //  seitsingerbob-hw1
4  //
5  //  Created by Robert Seitsinger on 8/31/15.
6  //  Copyright © 2015 cs378. All rights reserved.
7  //
8
9  import Foundation
10
11 func main()
12 {
13     // Write your highest level code here.
14     print("Hello, World!")
15 }
16
17 main()
18
```

Creating a Zip File on a Mac

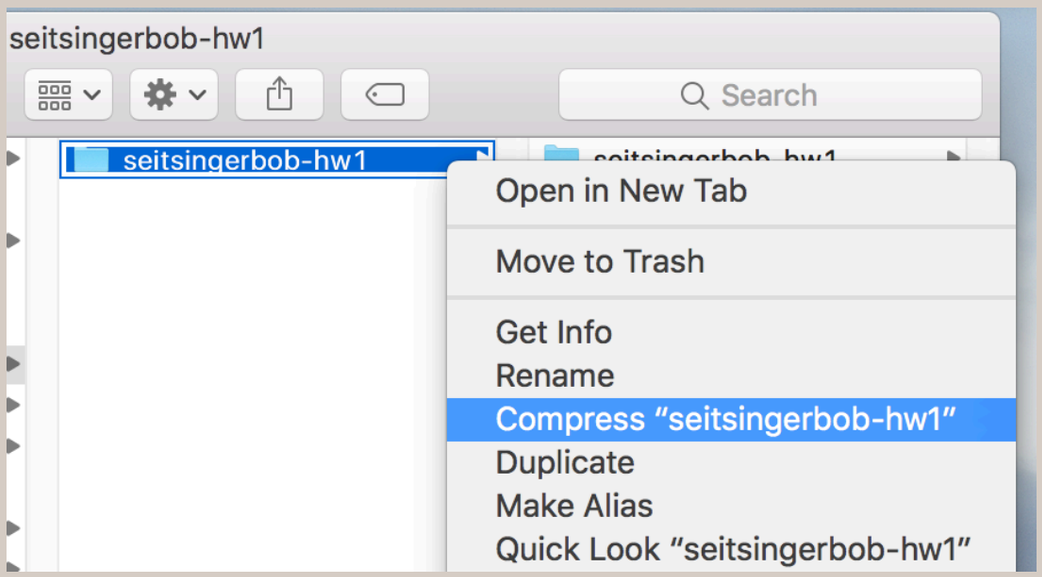
Creating a Zip File on A Mac

1. Single click to select the Xcode project folder.
2. Right mouse click the Xcode project folder and select “Compress <your-project-folder>”.

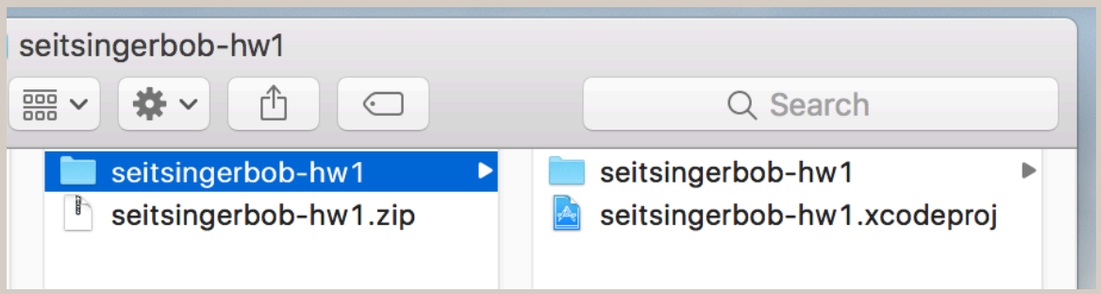
1



2



3



Homework 1

Homework 1

- Write a command line application in Swift
- Purpose is to start getting familiar with the Swift language before adding iOS to the mix
- Posted to Canvas
- Pay attention to the project and zip file naming requirements
- Have one week to complete