CSCI 4100 Laboratory Four

The Swift Programming Language

Introduction

In June 2014 Apple introduced the Swift programming language as an easier way to program iOS devices. This language has some similarities to C++ and Java, so it should be easier to learn than Objective-C, which was used before. At the present time there is not a lot of information on Swift. Probably the best source of information is Apple's web site: https://developer.apple.com/swift/resources/, which has two books on Swift, some videos and sample code. One of the interesting features of the latest version of XCode is playgrounds that give you an opportunity to interactively explore Swift without creating a complete project.

In this laboratory we will explore the Swift programming language, roughly following the "Swift Tour" in the "The Swift Programming Language" book. We will use playgrounds for this exploration. The "Swift Playgrounds" video provides much more information on playgrounds, which you might find useful when debugging Swift code. You should work through the examples as you read the lab.

Playgrounds

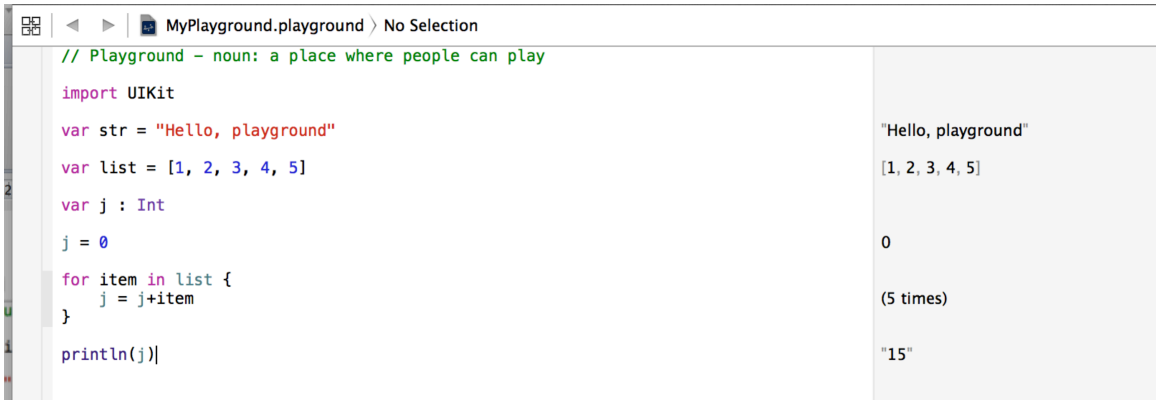
To get started start XCode and from the start screen select "Get Started With a Playground", see below:

In the next window you will be asked for the name of your playground and the platform, the default iOS is okay. Next you will be asked for a location for the playground, you can choose any convenient location.

Once you have done this you will have a playground like the one shown below. Note that it already has a hello greeting.

At the start the playground is divided into two areas. On the left you can type Swift statements and on the right the result of each statement is displayed. Each time you make a change to the swift code the complete set of statements will be run again. You can edit statements that you have already entered and view the results. This is a good way to experiment with the Swift language.

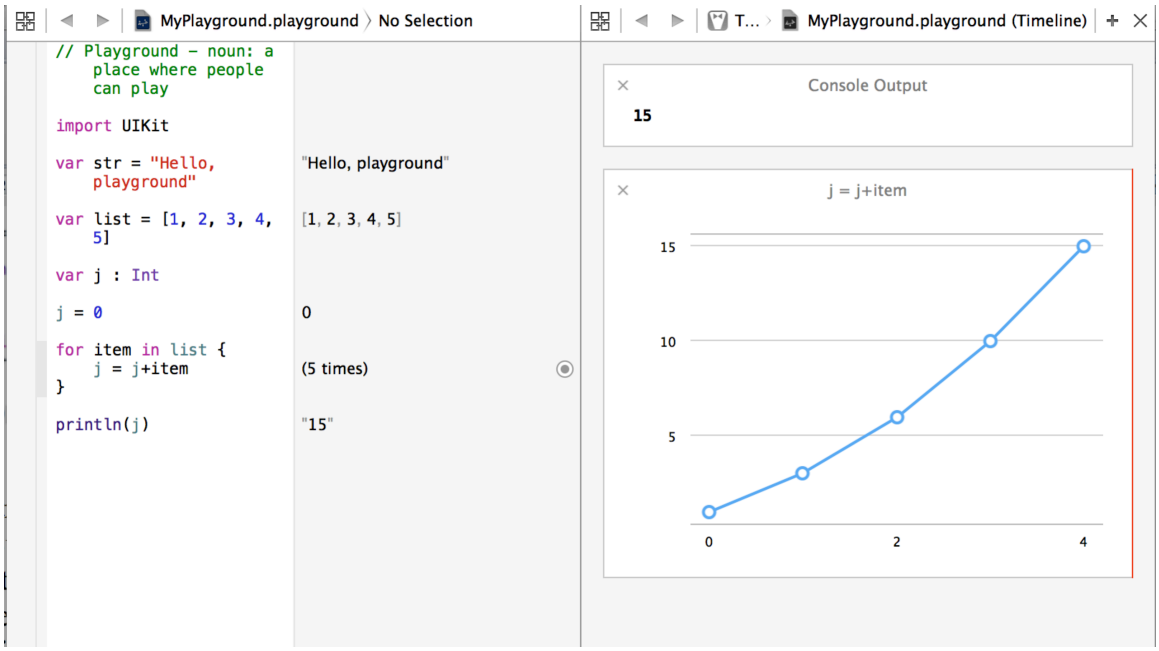
Now lets look at a small piece of Swift code:



Swift is a strongly typed language, every variable must have a type and this type can't be changed. But, Swift is also capable of inferring the types of variables. The declaration of a variable starts with "var" followed by the variable name. This can be followed by a ":" and the type of the variable, as in the declaration of "j". But, if an initial value is provided with the variable declaration Swift will infer the type of the variable from the type of the initial value. In this example "str" has the type String and "list" has the type array of Int. Also note that there are no ";" at the end of statements.

Now look at the for statement. To the right we see an indication that the for statement has been executed 5 times, but we don't know what happened inside of this loop. To

explore this mouse over the "(5 times)" and you will see a small circle appear at the end of the line. Click on this circle and you will get the history display for the for loop as shown below:

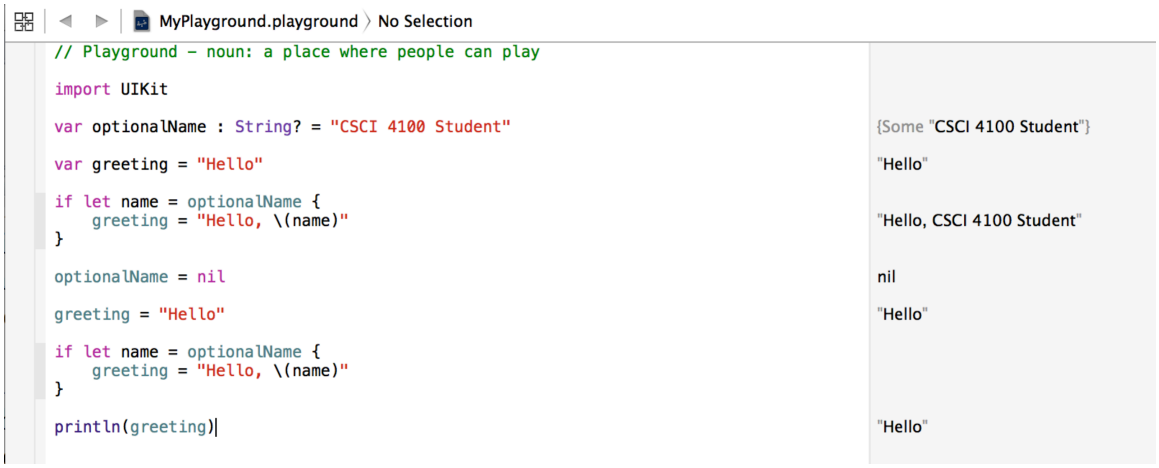


This shows how the value of "j" changes through each iteration of the loop. This is pretty obvious to us, but for more complex loops this can be an aid to debugging. This gives you an idea of how playgrounds work; it is now time to move on to Swift.

Swift

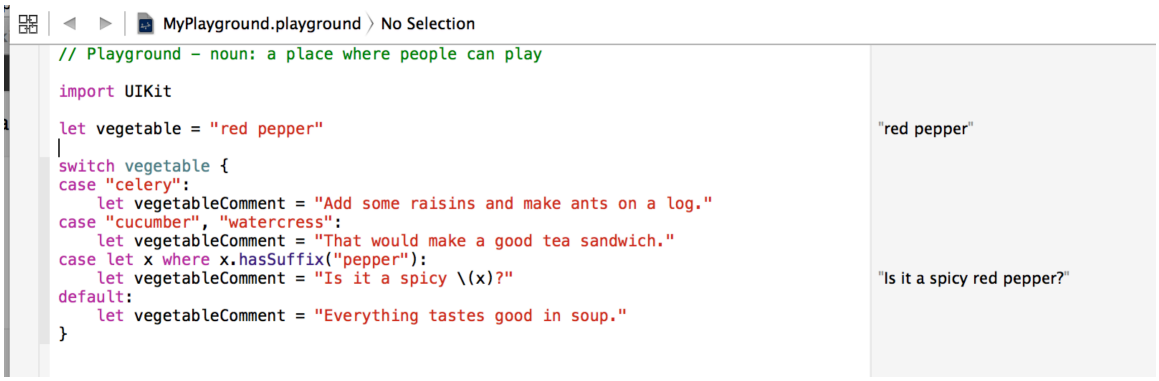
Swift has the standard types that we find in most programming languages including integers, floating point numbers and strings. Unlike other programming languages there is no implicit conversion between numeric types. For example, for example you cannot directly assign an integer to a floating point variable, you must do an explicit cast. The let statement can be used to declare constants; it has the same basic form as a variable declaration except it starts with the keyword "let".

Swift doesn't have pointer, but it must interface with a number of APIs that use pointers. To handle this Swift has a new set of types called optionals. An optional is like a normal variable, except it may not have a value. When it doesn't have a value you cannot reference the variable. Referencing an optional that doesn't have a value will result in a run time error. In order to safely reference an optional you must place it in an if statement and use a let as part of the condition to reference the value. An optional is indicated by placing a "?" at the end of the type name. A simple example of this is shown below:



The variable "optionalName" is an optional string variable. When we declare it we assign it an initial value. Later in the first if statement we test whether the variable has a value and if it does we assign that value to the variable "name" and execute the statements in the if statement. Next we assign nil to "optionalName" which indicates that the variable doesn't have a value. We then repeat the if statement, and in this case we notice that the statements inside the if statement aren't executed.

Most of the control structures in Swift are similar to the ones that you have seen in Java or C++. The one exception to this is the switch statement. An example showing the main features of the swift switch statement is:

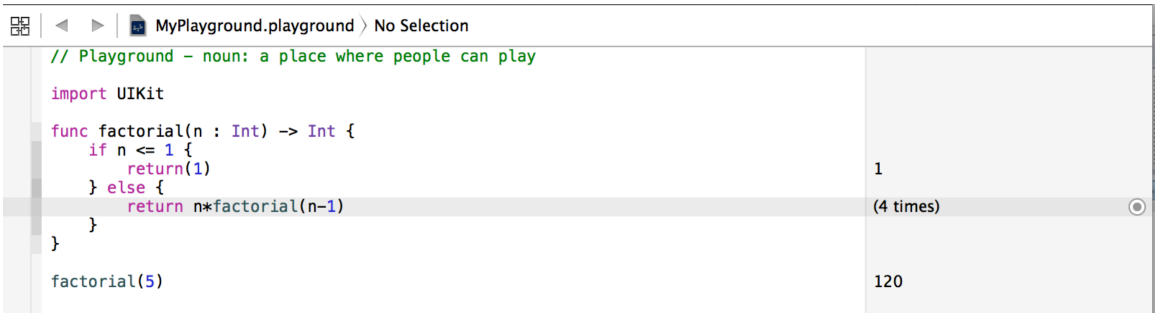


First of all notice that there are no break statements. A case ends just before the start of the next case. The "default" case is required in most switch statements, the only exception to this rule is enumeration types and there is a case for each value in the enumeration. The most interesting feature is that we can use statements as part of the case label. This is illustrated in the third case where a let statement is used to extract part of the variable. If the let statement is successful, the statements in the case are executed.

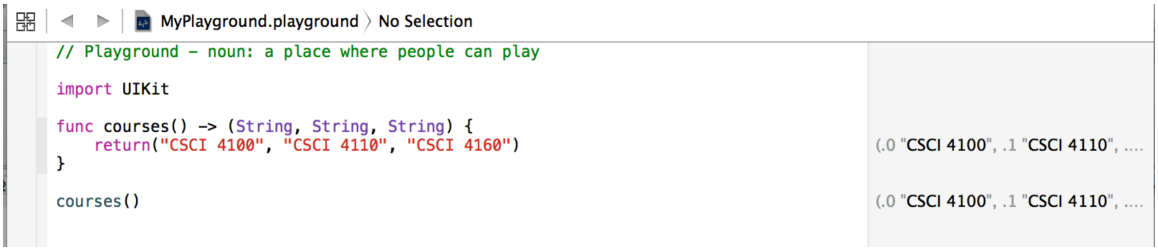
Functions and Closures

Function start with the keyword "func" followed by the name of the function. This is followed by the parameters to the function enclosed in "(" and ")". Finally there is a "->"

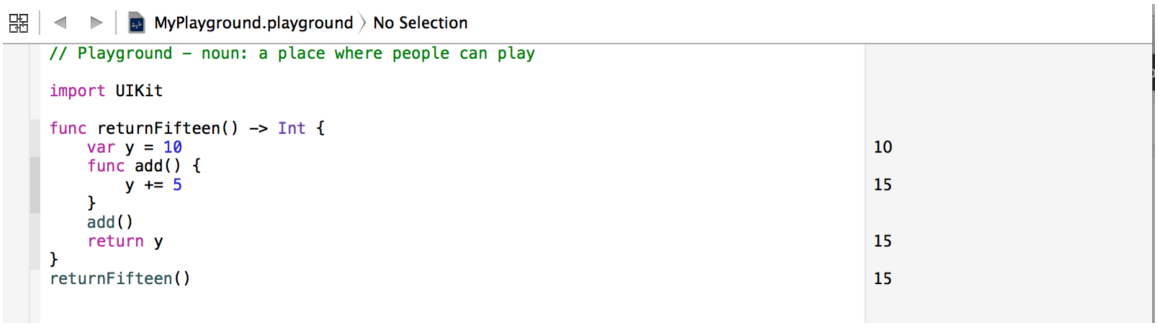
symbol followed by the type of the result. To illustrate the syntax we turn to our favorite function factorial:



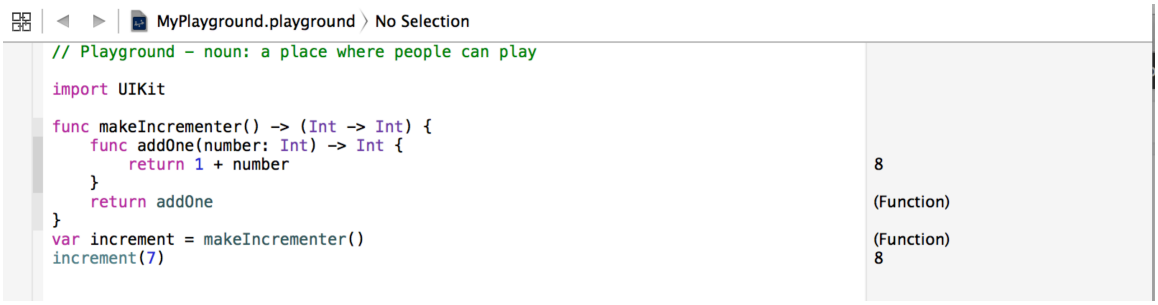
A function can return more than one value using a tuple, for example:



Functions can have other functions inside of them, for example:

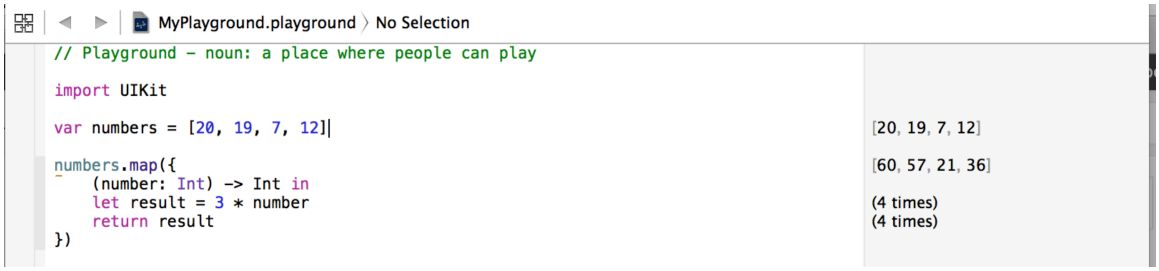


This can be useful for more complex functions where you might want to have a helper function, but you don't want that function to be visible to other functions in the program. Nesting functions inside other functions also allows us to return a function as a value. That is, functions are first class objects in Swift. They can be stored in variables and they can be returned by functions. The following Swift statements illustrate this:



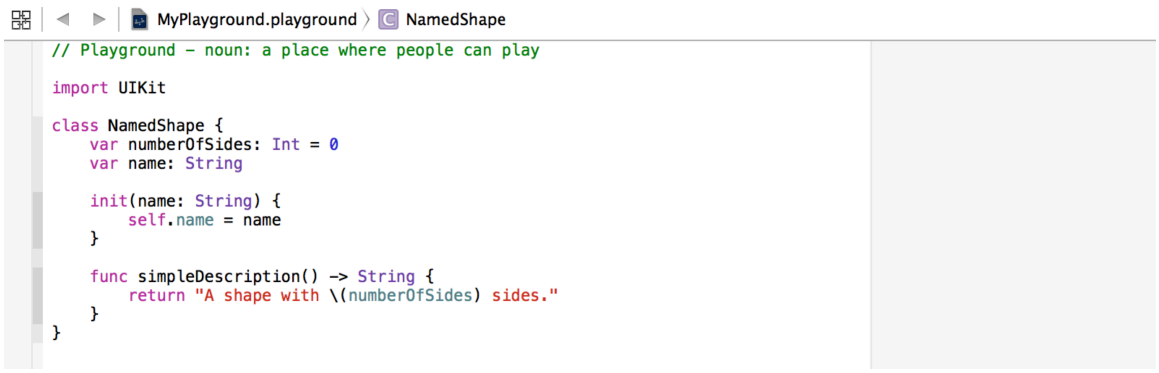
In this example the local function “addOne” is returned as the value of “makeIncrementer”. This function is stored in the variable “increment” and can be called through this variable.

In Swift we can have closures, functions without names. This can be done by writing the body of the function inside of “{“ and “}”. The following is an example of this:



Objects and Classes

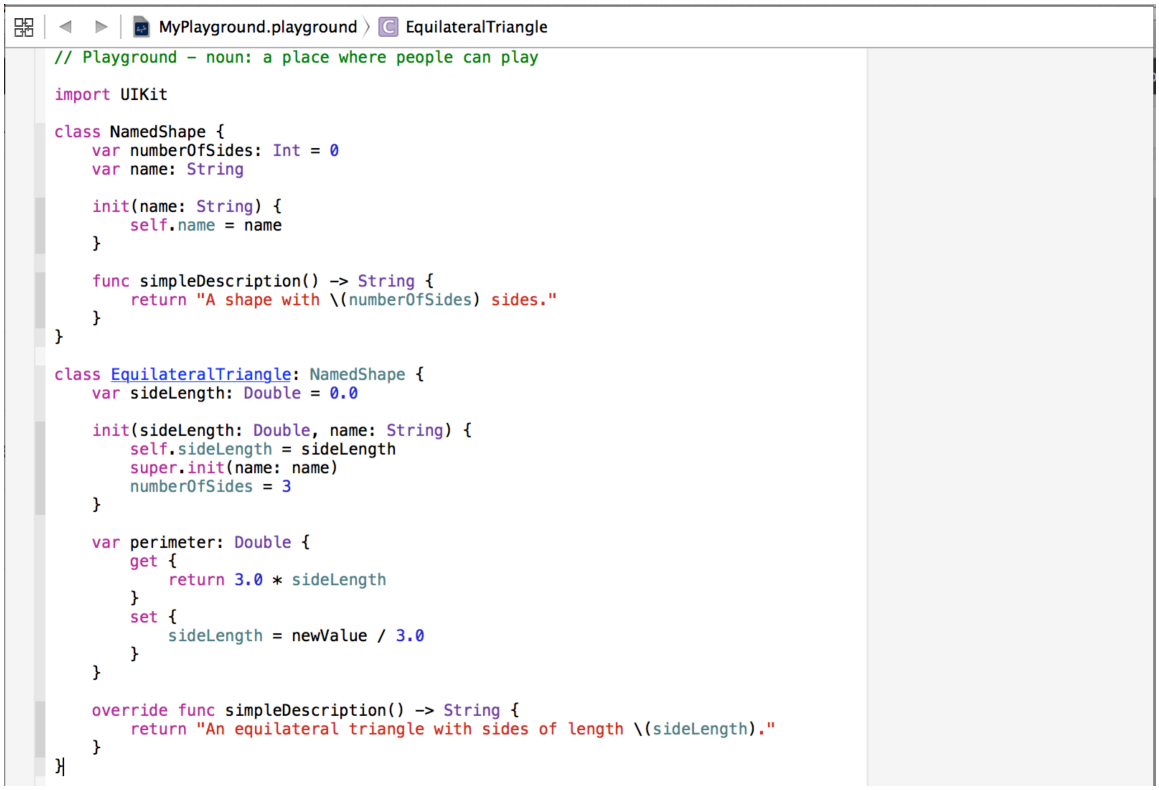
Classes in Swift are similar to classes in Java. The declaration and implementation of the class is in the same file; here is no need for a .h file for the class. We will illustrate classes with a few examples and explain the differences with Java. Our first example is:



The “init” function in a class is used to provide initialization for the class. There can also be a “deinit” function which is called before the object is destroyed giving you an opportunity to clean up resources allocated by the class. The variable “self” can be used to reference variables within the class, and the variable “super” can be used to reference the super class.

Inheritance is done in basically the same way as in Java. The “square” class below is an example of inheritance. Note the use of the “super” variable in the “init” procedure. The override keyword is required when you want to override one of the functions in the super class.


MyPlayground.playground 〉 No Selection
// Playground - noun: a place where people can play
import UIKit
class NamedShape {
var numberOfSides: Int = 0
var name: String
init(name: String) {
self.name = name
}
func simpleDescription() -> String {
return "A shape with \(numberOfSides) sides."
}
}
class Square: NamedShape {
var sideLength: Double
init(sideLength: Double, name: String) {
self.sideLength = sideLength
super.init(name: name)
numberOfSides = 4
}
func area() -> Double {
return sideLength * sideLength
}
override func simpleDescription() -> String {
return "A square with sides of length \(sideLength)."
}
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
test.simpleDescription()
27.04
"A square with sides of length 5.2."
{{numberOfSides 4 name "my test...
27.04
"A square with sides of length 5.2."


MyPlayground.playground 〉 EquilateralTriangle
// Playground - noun: a place where people can play
import UIKit
class NamedShape {
var numberOfSides: Int = 0
var name: String
init(name: String) {
self.name = name
}
func simpleDescription() -> String {
return "A shape with \(numberOfSides) sides."
}
}
class EquilateralTriangle: NamedShape {
var sideLength: Double = 0.0
init(sideLength: Double, name: String) {
self.sideLength = sideLength
super.init(name: name)
numberOfSides = 3
}
var perimeter: Double {
get {
return 3.0 * sideLength
}
set {
sideLength = newValue / 3.0
}
}
override func simpleDescription() -> String {
return "An equilateral triangle with sides of length \(sideLength)."
}
}

The EquilateralTriangle class shows how we can have custom setter and getter procedures for the properties in our class. These functions are called when the variable is referenced or a new value is assigned to the variable.

Laboratory Exercise

Work through the "Enumerations and Structures" sections of the Swift Tour part of the "The Swift Programming Language" book on your own. Show the TA the results of your work.