# LO4 Object oriented Concepts

# Functions

Functions use the keyword func. The syntax for declaring a function is:

func myFunction( parameterList ) -> returnType

{

//function code

return value

}

Where parameterList is a comma separated list of parameterName : parameterType

For example:

func sumDigits( number: Int ) -> Int {

var result = 0

//parameters are constants(let) since I need to change

//number use a variable instead

var numb = number

while( numb > 0 )

{

result += numb % 10

numb = numb / 10

}

return result

}

When calling a method you must include the parameter name before the value for that parameter.

print( sumDigits( number : 141 ) )

var res = sumDigits( number : 2002 )

print( res )

If a function does not return a value just omit the return type complete as well as the return statement.

func displayHello() {

println( "Hello" )

}

displayHello()

## Terminology – method vs function

Method and functions are really the same thing. However, Swift uses the term method when defined inside of a class and the term function when defined outside of a class.

## External Parameter Names

Parameters can have different internal and external names. Internal names are used within the body of the function to access the parameter. External parameter names are used when the function is called.

Without external parameter names:

func determineArea( width: Int, height : Int ) -> Int

{

return width \* height

}

println( determineArea( 20, 40 ))

With external parameter names. You must specify the parameter names when calling the function. This makes the code more self documenting.

func calcArea ( width w : Int, height h : Int ) -> Int

{

return w \* h

}

print( calcArea( width: 10, height: 5 ) )

## Default Values

A function may have default values for parameters – in which case it will be optional to provide the parameter when the function is called. An external name MUST be used with parameters that have a default value – the external parameter name is assumed to be the same as the internal parameter name unless an explicity external parameter name is specified.

func showMessage( message : String, times : Int = 1 )

{

for \_ in 1...times

{

print( message )

}

}

showMessage(message : "Hello")

showMessage( message: "Goodbye", times : 5)

There can be multiple, or even all, parameters with default values.

func showMessage( message : String = "Test", times : Int = 1, uppercase : Bool = false )

{

for \_ in 1...times

{

if uppercase

{

print( message.uppercased() )

}

else

{

print( message )

}

}

}

showMessage( message: "Hello")

showMessage( uppercase: true )

//Note: parameter name must be specified

showMessage( message:"Goodbye", times: 10 )

Exercise caution with overloaded functions – a method taking in a String and a second parameter with a default value means it will match a function call which just takes in one String parameter. Although, generally overloaded functions are less necessary (but not completely) with the capability to have a default value values.

**Student Practice:** Write a method to calculate the total price of an item(s). Parameters should be price(required), quantity(optional, default to 1), taxRate(optional, default to 0). Use external parameter names as: price, quantity and taxRate. But use internal parameter names of price, qty, rate. Method name is calculatePrice. Method should return an optional Double value which is the calculated total – price \* quantity \* ( 1 + taxRate) OR nil if qty, taxRate or price is less than 0.

//Don’t include external label if same as internal label

func calculatePrice( price : Double, quantity qty: Int = 1, taxRate rate : Double = 0 ) -> Double?

{

var result : Double?

if rate < 0 || qty < 0 || price < 0

{

result = nil

}

else

{

result = Double( price ) \* Double( qty ) \* ( 1 + rate )

}

return result

}

var amt = calculatePrice( price : 5.99 )

print( amt! )

amt = calculatePrice( price : 10, quantity : 20 )

print( amt! )

amt = calculatePrice( price : 20, quantity: 5, taxRate : 0.1 )

print( amt! )

amt = calculatePrice( price : 20, quantity: -10, taxRate : 0.1 )

print( amt as Any)

amt = calculatePrice( price : -5, quantity: 5, taxRate : 0.1 )

print( amt as Any)

amt = calculatePrice( price : 20, quantity: 5, taxRate : -0.1 )

print( amt as Any)

## Variadic Parameters

A varaidic parameter allows 0, 1 or more parameters of the same type to be passed to a function. Inside the function the parameters are treated as a single array. The parameters are stored in the array in the order they are provided in the method call. A function can be called with any number of parameters.

A function with a variadic parameter can have other parameters as well but only one variadic parameter and the variadic parameter must be the last parameter in the list.

func findLargest( numbers: Int... ) ->Int

{

var largest = Int.min

//Since variadic parameter numbers is an array

for numb in numbers

{

if numb > largest

{

largest = numb

}

}

return largest

}

let largest = findLargest( numbers: 12 )

print( largest )

print( findLargest( numbers: 45, 2, 198766, 232312, -56, -299999))

print( findLargest( numbers: -8000, -90, -12 ))

**Practice:** Write a function that takes a one Int parameter and one variadic parameter. The first is the number to find and the second is the list of numbers. If the number to find is in the list of numbers then return true, otherwise return false.

func findNumber( numberToFind: Int, numbers : Int... ) -> Bool

{

for numb in numbers

{

if numb == numberToFind

{

return true

}

}

return false

}

var result = findNumber( numberToFind: 12, numbers: 34,45,65,4343,12 )

print( result )

result = findNumber( numberToFind: 12, numbers: 34,45,65,4343,78 )

print( result )

## Parameters are defined as Let

By default in Swift all parameters are call by value. In fact there are defined using let rather than var so they cannot be modified within the function itself. (Causes a syntax error if you try to modify)

Parameters could be marked with var so the parameter can change in the method but this was deprecated in 3.0

## Modifying Function Parameters – using inout

Swift’s default way of handling parameters is designed to be safe and is considered best practice. However, there are some cases where modifying the data that is passed into a function is necessary. This can be done by marking the parameter as inout which indicates the method is intended to change the value of the data.

For example, perhaps when we divide a number in half we always want to take the bigger half if the original number was odd. We could write a function to do this that has an inout parameter. Note: when calling the method you must specify & before the variable to indicate it is a inout parameter, this makes it clear to the program that the variable will be modified in the function. Of course this means you cannot pass let variables to a method that is using an inout parameter.

func divideInHalfSpecial( number : inout Int )

{

if number % 2 == 1

{

number = number / 2 + 1

}

else

{

number = number / 2

}

}

var val1 = 12

divideInHalfSpecial( number: &val1 )

print( val1 )

var val2 = 7

divideInHalfSpecial( number: &val2 )

print( val2 )

let val3 = 11

//Syntax error

divideInHalfSpecial( number: &val3 )

**Practice:** Write a swap method that takes two Int parameters, it must swap the value of the two parameters.

func swap( numb1 : inout Int, numb2 : inout Int )

{

let temp = numb1

numb1 = numb2

numb2 = temp

}

var val1 = 23

var val2 = -2

swap( numb1: &val1, numb2: &val2 )

print( val1 )

print( val2 )

## Object References as Parameters

Just like in Java – if you pass an object reference as a parameter to a function – the address of the object a let variable – You cannot change the object reference address. But you can change the underlying object so you can still have privacy leaks in Swift.

//More on this when talking about classes

CODE CHECKED TO HERE!!!

## Function Types

A function is an unnamed type, but every function is explicitly typed. The type of the function is defined by the number and types of its parameters and the return type of the function. So function type can be defined as:

( Int, Int ) -> Int

Insert appropriate data types where needed. You can also define a function that retrurns nothing:

( Double, String ) ()

Since a function can be defined in this way we can create a variable that is a function type:

Var f : ( Int, Int) ->Int

Or we can pass a function into a function as a parameter. Or a function can be returned from function.

Example:

func multiply( left : Int, right : Int ) -> Int {

return left \* right

}

func sum( left : Int, right : Int ) -> Int {

return left + right

}

func divide( left : Int, right : Int ) -> Int

{

return left / right

}

func subtract( left : Int, right : Int ) -> Int {

return left - right

}

func pickOperation( op : String ) -> (Int, Int)->Int {

if op == "+"

{

return sum

}

else if op == "-"

{

return subtract

}

else if op == "/"

{

return divide

}

else

{

return multiply

}

}

var operation : String

operation = "+"

var f = pickOperation( op : operation )

//Does add

print( f( 4, 5 ) )

operation = "\*"

f = pickOperation( op : operation )

//does multiply

print( f( 4, 5 ) )

operation = "/"

f = pickOperation( op : operation )

//Does divide - truncate result(integer division like Java)

print( f( 27, 10 ) )

operation = "-"

f = pickOperation( op : operation )

//subtract

print( f( 4, 5 ) )

**Student Practice:** Create two functions – one called welcomeEnglish which takes in a name as a parameter and returns a String “Hello *name*”, a second function should be called welcomeFrench which takes in a name as a parameter and returns a String “Bonjour *name”.* Then create a function type variable that represents a function with one String parameter that returns a String. Set this variable to each of the methods in turn and print the result of calling the method.

func welcomeEnglish( name : String ) -> String

{

return "Hello \(name)"

}

func welcomeFrench( name : String ) -> String

{

return "Bonjour \(name)"

}

var greeting : (String) -> String

greeting = welcomeEnglish

print( greeting("Sharon") )

greeting = welcomeFrench

print( greeting("Sharon") )

# Struct

A structure is a collection of members. Structures are used to encapsulate information. Structures and classes are very similar but the key difference is that structures are value types and classes are reference types. This means structures are copied when they are passed to something like a function.

A structure is made up of properties and methods. Properties can be regular properties or computed properties.   
Example:

public struct Room{

public var length : Int

public var width: Int

//computed property - floor area

public var area: Int { return length \* width }

public func displayRoomInfo() {

print( "Room is \(length) X \(width)")

print("with an area of \(area)")

}

}

var bedroom : Room = Room( length: 20, width: 10 )

var bathroom : Room = Room( length: 10, width: 4 )

print( bedroom.area )

print( bathroom.area )

bedroom.displayRoomInfo()

**Practice:** Write a structure that represents a fraction. It has a numerator and demoninator properties(Int). Give a default value of 1 to numerator and denominator. Have a computed property called toDouble which returns the double equivalent of the fraction. Have a computed property, description, that creates a String representation of the fraction. For example, if numerator is 1 and demoninator is 2 then the double value is .5. Also, have a method, isValid, that returns a boolean value indicating if it is a proper fraction. A proper fraction is 1 or less – ie. The numerator is NOT greater than the denominator. Then try to create a half and three eights fraction objects.

public struct Fraction

{

//Note: Must be public to get default initializer(private need to create memberwise initializer)

public var numerator : Int

public var denominator : Int

public var description: String {

return "Fraction is \(numerator)/\(denominator)"

}

public var toDouble : Double {

return Double( numerator) / Double( denominator)

}

public func isValid () -> Bool

{

return numerator <= denominator

}

}

var half : Fraction = Fraction( numerator: 1, denominator: 2 )

print( half.description )

print( half.isValid() )

print( half.toDouble )

var threeEights = Fraction( numerator: 3, denominator: 8 )

print( threeEights.description )

print( threeEights.isValid() )

print( threeEights.toDouble )

## Initializers( constructors )

Provided no other initializers are created a structure always have member wise initializer and if properties have default value then it will also have a default initializer(constructor) and a member wise initializer( a constructor that takes all members of the struct)

A memberwise initializer: takes in each property as an argument. Names of each parameter must be specified.

var bedroom : Room = Room( length: 20, width: 10 )

If no initializer are defined AND the properties have default values then a default initializer will also be available. Add default values to room dimensions. Then show calling default constructor.

If we create an initializer in the struct then default and memberwise initializer are not available. Try adding:

public init( length: Int, width : Int )

{

self.length = length

self.width = width

}

If we added our own initializer and we still want default and/or memberwise initializer then we have to add them ourselves.

## Mutating methods

By default properties cannot be modified within methods of structs. If a method needs to modify a property in a struct it must be marked as mutating.

public struct Room{

public var length : Int

public var width: Int

//computed property - floor area

public var area: Int { return length \* width }

public func displayRoomInfo() {

print( "Room is \(length) X \(width)")

print("with an area of \(area)")

}

public mutating func setLength( newLength : Int ) {

length = newLength

}

public mutating func setWidth( width : Int )

{

self.width = width

}

}

var bedroom : Room = Room( length: 20, width: 10 )

var bathroom : Room = Room( length: 10, width: 4 )

print( bedroom.area )

print( bathroom.area )

bedroom.displayRoomInfo()

bedroom.setWidth( width : 25 )

bedroom.setLength( newLength : 12 )

bedroom.displayRoomInfo()

**Student Practice:**  Write a function to reduce the fraction to lowest possible fraction in the Fraction class. For example, if the fraction was 3/12 then the reduce method would change the fraction to ¼. OR 4/12 would become 1/3(not 2/6)

public mutating func reduce()

{

//Reduce fraction to smallest

//Find highest common denominator

var value = numerator < denominator ? numerator : denominator

while value > 0

{

if numerator % value == 0 && denominator % value == 0

{

numerator = numerator / value

denominator = denominator / value

//done

value = 0

}

}

}

## Structs are value types

A structure is a value type so if a structure is passed as a parameter( parameters are let!) you cannot call mutating methods on that structure.

## Type Properties(Static Properties)

Type properties allow data to be shared between all instances. Instance properties have one value per instance and type properties have one value shared by all instances. Use the modifier static.

## Type Methods( Static methods)

Static methods are called against the struct. Cannot refer to instance properties but can refer to type properties.

public struct Fraction

{

private var numerator : Int = 1

private var denominator : Int = 1

public static var item : Int = 10

public var description: String {

return "Fraction is \(numerator)/\(denominator)"

}

public var toDouble : Double {

return Double( numerator) / Double( denominator)

}

static func setItem( item : Int ) {

self.item = item

}

public func isValid () -> Bool

{

return numerator <= denominator

}

mutating func setNumerator( numerator : Int ) {

self.numerator = numerator

}

mutating func setDenominator( denominator : Int ) {

self.denominator = denominator

}

}

//Static properties must use struct name

print( Fraction.item )

Fraction.setItem( item : 22 )

print( Fraction.item )

# Classes

## Properties

Class have instance properties and computed properties. Specify modifer for the property. Generally a good idea to make instance properties private. Computed properties would generally be public. Use let for constants and var for variables. According to CST coding standard let variables should be all capitals – this does not seem to be a convention in Swift.

public class Product

{

private var name : String

private var price : Double

public let TAX\_RATE : Double = 0.05

public var description : String {

return "\(name) costs $\(price)"

}

}

## Initializers

Initializers are the methods called when an object is created(Constructors in Java). Initializers are usually responsible for ensuring that properties have an initial value. If all instance properties have default values then it is not required to explicitly declare an initializer as a default one is automatically created. If properties do not have default values then you MUST declare an initializer(constructor).

To create an initializer declare a function with no return type and no func keyword and uses the name init.

public init( name : String, price : Double ) {

self.name = name

self.price = price

}

Designated initializer – the initializer that does the real work. Initializers can be overloaded – initializers that call the designated initializer are called convenience initializers. Note: if you add any initializers no default initializer is created so if you still want the default initializer you need to explicity create it.

An initializer that calls another initializer is called a convenience initializer. You must use the keyword convenience on convenience initilizers. Unlike Java the call to another initializer within a convenience initializer does not need to be the first statement – you can do other stuff first.(In example below try adding println(“Try This”) before self.init)

//Convenience Initializer

public convenience init( name : String )

{

self.init( name: name, price : 0.0 )

}

You can have any number of designated initializers and any number of convenience initializers.

## Functions

Add function

public func calculateCost( quantity : Int ) -> Double

{

var total : Double = Double( quantity ) \* price

//Apply tax rate

total = total + total \* TAX\_RATE

return total

}

## Inheritance

Derive a new class based on an existing class. Create Book that is a subclass of Product. Objects of type Book inherit any public properties and/or methods from Product. Within the code of the subclass you can access private instance properties. (Different from what Pete said in course??) Just like in Java, in Swift you can inherit or extend one class only. But of course multiple levels of inheritance. To inherit in Swift follow class declaration with : and parent name.

public class Book : Product

{

}

Methods that override methods in the parent class must be designated with the override keyword. Can override properties as well.

Using attributes in child must have public modifier – in Swift 3.0 not Swift 2.0. Can control by providing set/get methods in parent – not worth it??

public class Product

{

public var name : String

public var price : Double

public let TAX\_RATE : Double = 0.05

public var description : String {

return "\(name) costs $\(price)"

}

//Designated initializer

public init( name : String, price : Double )

{

self.name = name

self.price = price

}

//Convenience Initializer

public convenience init( name : String )

{

print( "Try This")

self.init( name: name, price : 0.0 )

}

public func calculateCost( quantity : Int ) -> Double

{

var total : Double = Double( quantity ) \* price

//Apply tax rate

total = total + total \* TAX\_RATE

return total

}

}

var p : Product = Product(name: "Saw")

public class Book : Product

{

public override var description : String {

return "The book, \(name) costs $\(price)"

}

public override func calculateCost( quantity : Int ) -> Double

{

return price \* Double( quantity )

}

}

var b : Book = Book( name: "Java Programming", price: 69.99 )

print( b.description )

print( b.calculateCost( quantity : 1 ) )

You can call parent’s property or method using super keyword(like in Java)

public override var description : String {

return "The book " + super.description

}

## Polymorphism

Polymorphism is the idea that an object can take on many forms. A Book is a book AND it is a Product. This means we can hold a Book object in an array of Product objects. Dynamically at runtime it determines which version of a method or property to use based on object that was actually instantiated. For example:

var prods : [Product] = [Product] ()

prods.append( p )

prods.append( b )

prods.append( Book( name: "Swift is Awesome", price : 49.99 ) )

for temp in prods

{

print( temp.description )

}

## Initializer in Subclasses

An initializer in a subclass must call the designated initializer in the super class. (If no initializers are provided in the child class then parent initializers are called passed on arguments provided).

But you can additionally have initializers in child class as well. (And are usually required. ) As soon as an initializer is added to the child class then the initializers in parent class are not accessible from child object.(Need to call child initializer).

If you create an initializer in the child class that takes in the same type and number of parameters as the parent you must use the override keyword.

Within the initializer call the parent’s initializer by using keyword super.

public class Book : Product

{

public override var description : String {

return "The book, \(name) costs $\(price)"

}

public override init( name: String, price : Double )

{

super.init( name : name, price : price)

print("In child initializer")

}

public override func calculateCost( quantity : Int ) -> Double

{

return price \* Double( quantity )

}

}

**Student Practice:** Create an Account class. The withdraw method should only withdraw if there is enough money left in the account.(Balance cannot go below zero) Return true if withdraw was successful, false otherwise. Description is a computed property – return a String containing account id has balance. For example “1001 has a balance of $1000”

|  |
| --- |
| Account |
| -accountId: Int  -balance: Double  +description: String |
| +init( accountId : Int, balance : Double )  +withdraw ( amount : Double ): boolean  +deposit ( amount : Double ): void |

Create a ChequingAccount class that inherits from Account and has an attribute called overdraftLimit. This class should override the withdraw method to take overdraft limit into account.(Overdraft limit allows the account to go negative by up to the overdraft limit so if the account has an overdraft limit of 1000 then the balance can go as low as -1000). Also, override description computed property to include overdraft limit in string – for example: “1001 has a balance of $1000 and an overdraft limit of $500”

|  |
| --- |
| ChequingAccount |
| -overdraftLimit : float  +description: String |
| +init( accountId : Int, balance : Double, overdraftLimit : Double )  +withdraw ( amount : Double ): boolean |

Try creating a couple of Account and ChequingAccount objects and test out the methods.

public class Account

{

var accountId : Int = 1000

var balance : Double = 0

public var description : String {

return "\(accountId) has a balance of $\(balance)"

}

public init( accountId : Int, balance : Double )

{

self.accountId = accountId

self.balance = balance

}

public func deposit( amount : Double )

{

balance += amount

}

public func withdraw( amount : Double ) -> Bool

{

var result : Bool

result = false

if ( balance >= amount )

{

result = true

balance -= amount

}

return result

}

}

var a : Account = Account( accountId : 2001, balance: 50.0)

print( a.description )

print( a.withdraw( amount : 100 ) )

print( a.description )

print( a.withdraw( amount : 20 ) )

print( a.description )

public class ChequingAccount : Account

{

private var overdraftLimit : Double = 500

public override var description : String {

return "\(accountId) has a balance of $\(balance) and " +

"overdraft limit of $\(overdraftLimit)"

}

public init(accountId : Int, balance : Double, overdraftLimit: Double )

{

// can be in front of or after call to parent's constructor

// self.overdraftLimit = overdraftLimit

super.init( accountId : accountId, balance : balance )

self.overdraftLimit = overdraftLimit

}

public override func withdraw( amount : Double ) -> Bool

{

var result : Bool

result = false

if ( balance + overdraftLimit >= amount )

{

result = true

balance -= amount

}

return result

}

}

var c : ChequingAccount = ChequingAccount( accountId: 7077, balance: 1000, overdraftLimit : 1500 )

print( c.withdraw( amount : 2000 ) )

print( c.description )

## Computed Properties and get/set

Computed properties are a way to make a property publicly modifiable but in a controlled manner – can ensure that it is not set to invalid value.

public class Circle

{

var radius : Double = 0

var diameter : Double

{

//newValue refers to the value diameter was set to from outside class

set {

if newValue > 0

{

radius = newValue / 2

}

}

get {

return radius \* 2

}

}

public var description : String {

return "Circle has a radius of \(radius)"

}

//No initializer needed since type property has default value

}

var c : Circle = Circle()

print( c.description )

c.diameter = 10 //uses Set

print( c.description )

print( "Diameter is " )

print( c.diameter ) //uses get

c.diameter = -1

print( c.description ) //Not changed

## Abstract classes

Swift does not have abstract classes. We need to use protcols to mimic abstract class idea.

## Protocols

Swift has protocols which are the same thing as interfaces in Java. A class may implement any number of interfaces. A protocol( interface) is a collection of specifications without implementations. Any class that implements a protocol is required to implement the specifications. CustomStringConvertible is a protocol defined in Swift that requires a class to have a description property.

### public class MyClass : CustomStringConvertible

### {

### private var number : Int = 10

### 

### public var description : String {

### return "Number is \(number)"

### }

### 

### }

### public class AnotherClass : CustomStringConvertible

### {

### public var description : String

### {

### return "This class is weird - no attributes"

### }

### }

### var test1 : MyClass = MyClass()

### var test2 : AnotherClass = AnotherClass()

### var array : [CustomStringConvertible] = [CustomStringConvertible]()

### array.append( test1 )

### array.append( test2 )

### for p in array

### {

### //According to Peter’s notes we should be able to do println( p )

### //And it does work in XCode

### print( p.description )

### }

### Creating Protcol

To create a protocol use the keyword protocol instead of class. Common convention in Swift is to have protocols have an “able” suffix – CustomStringConvertible, Taxable, etc. Protocols can have methods and computed properties.

public protocol Displayable

{

//Note: Computer property did not work in browswer but does work in // XCode - in browser just keeps complaining that class does not match

// protocol

// var fullDescription : String (get)

func displayDetails()

}

public class MyClass : Displayable

{

private var number : Int = 10

public var fullDescription : String {

get {

return "Number: \(number)"

}

}

public func displayDetails()

{

print("My number is \(number)")

}

}

var test : Displayable = MyClass()

test.displayDetails()

# Differences between Struct and Class

Struct is a value type whereas class is a reference type

Struct and Class both get a default initializer if no other initializers are defined (and properties have default values ). Struct also gets a memberwise initializer.

The properties of a const struct (defined with let) cannot be modified. The properties of a const class can be modified.

class TestClass

{

var number : Int = 10

var description : String { return "Number: \(number)" }

}

struct TestStruct

{

var name : String = "Sharon"

var description : String { return "Name: \(name)" }

}

var varClass : TestClass = TestClass()

var varStruct : TestStruct = TestStruct()

varClass.number = 12

print( varClass.description )

varStruct.name = "Shane"

print( varStruct.description )

let letClass : TestClass = TestClass()

let letStruct : TestStruct = TestStruct()

letClass.number = 5

//This gives syntax error

letStruct.name = "Broomhilda"

print( letClass.description )

print( letStruct.description )

## Struct is a value type and Class is a reference

This means when passed to a method – contents of class may change but contents of struct will not. Consider this example:

class TestClass

{

var number : Int = 10

var description : String { return "Number: \(number)" }

}

struct TestStruct

{

var name : String = "Sharon"

var description : String { return "Name: \(name)" }

}

func changeClass( inClass : TestClass )

{

inClass.number = 42

}

func changeStruct( inStruct : TestStruct )

{

//syntax error

// inStruct.name = "Heathcliff"

}

var varClass : TestClass = TestClass()

var varStruct : TestStruct = TestStruct()

changeClass( inClass : varClass )

print( varClass.description )

changeStruct( inStruct : varStruct )

print( varStruct.description )

If you really want the contents of the original structure to change you can always pass the structure as an inout parameter and then the original structure will be modified. For example:

func changeStruct( inStruct : inout TestStruct )

{

inStruct.name = "Heathcliff"

}

changeStruct( &varStruct )

println( varStruct.description )

# Enumerations

An enumeration is a collection of items. It is complete, no items can be removed or added. An enumeration in Swift is a value type. For example:

enum Weekday

{

case Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

}

var day = Weekday.Monday

if day == Weekday.Monday

{

println("It is Monday!")

}

**Student Practice:**Write a enumeration to hold a size. Sizes can be small, medium and large.

enum Size

{

case Small, Medium, Large

}

## Raw Types

Each enumeration can be mapped onto a specific type which is referred to as the raw type of the enumeration. Raw types should only be used when they are useful. An example of when this might be useful if you have an enumeration for months and you want to map each month to a Int value so you can index into an array.

It is possible to convert back and forth with raw types.

## enum Weekday : Int

## {

## case Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

## }

## var day = Weekday.Wednesday

## let dayNum = day.rawValue

## print( dayNum )

## var thurs = Weekday( rawValue:4 )

## //What do you think thurs holds?

## print( thurs as Any )

## //It is an optional value - it may be nil if raw value was

## //not valid for the enumeration

## var something = Weekday( rawValue: 11 )

## print( something as Any )

## //forced unwrapping

## if thurs != nil

## {

## let thursday = thurs! //Note exclamation point

## print( "Unwrapped thursday!")

## }

## Switch Statements and Enumerations

Switch statements are very useful with enumerations.

var day = Weekday.Wednesday

switch day {

case .Monday : print("It is Monday!")

case .Tuesday : print("It is Tuesday!")

case .Wednesday : print( "It is Wednesday!")

case .Thursday : print("It is Thursday!")

case .Friday : print("It is Friday!")

case .Saturday : print( "It is Saturday!")

case .Sunday : print( "It is Sunday")

}

What happens if day is optional type? “var day = Weekday( rawValue : 6 )” causes an error. Instead need to force unwrapping:

var dayMaybe = Weekday( rawValue : 6 )

let day = dayMaybe!

Keep in mind that if dayMaybe was nil then the let day = dayMaybe! would be an error.

## Enumerations can have Methods

You can add methods to enumerations. For example we could provide a method to return the String representation of the enumeration.

enum Weekday : Int

{

case Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

func description() -> String

{

var result : String = ""

switch self {

case .Monday : result = "Monday"

case .Tuesday : result = "Tuesday"

case .Wednesday : result = "Wednesday"

case .Thursday : result = "Thursday"

case .Friday : result = "Friday"

case .Saturday : result = "Saturday"

case .Sunday : result = "Sunday"

}

return result

}

}

var day = Weekday.Thursday

print( day.description() )

Could also add a method to determine next day after this one

func nextDay() -> Weekday {

var nextDay : Int = self.rawValue

nextDay = nextDay % 7 + 1

//Note use of ! to force unwrapping

return Weekday( rawValue : nextDay )!

}

var day = Weekday.Thursday

print( day.description() )

var nextDay = day.nextDay()

print( nextDay.description() )

**Student Practice:**  Change size enumeration to have a raw value. Add a description method. Add a goSmaller and a goLarger methods – these move one size smaller or one size larger. If already at the smallest or largest just stay there. Try these methods two ways – first just return the new smaller size. Then try to write it by changing this enumeration itself.

enum Size : Int

{

case Small = 1, Medium, Large

func description() -> String

{

var result : String = ""

switch self

{

case .Small : result = "Small"

case .Medium : result = "Medium"

case .Large : result = "Large"

}

return result

}

mutating func goSmaller()

{

if self == .Medium

{

self = .Small

}

else if self == .Large

{

self = .Medium

}

}

//This method returns a new Size enumeration

func goLarger() -> Size

{

if self == .Small

{

return .Medium

}

else if self == .Medium

{

return .Large

}

return .Large

}

}

var size = Size.Medium

print( size.description() )

size.goSmaller()

print( size.description() )

var bigger = size.goLarger()

print( bigger.description() )

# Arrays in Swift

Arrays are named types. In Swift Collections are ALWAYS explicitly typed.(Generics is forced, not optional)

## Declaring an array

Must specify type when array is specified. Following creates an array of size 0.

//Short form

var myArray : [Int] = [Int] ()

//Full Form

var myArray : Array<Int> = Array<Int>()

Arrays can be initialized with an array literal:

var numbers = [ 10, 20, 30, 40, 50 ]

var fruits = ["apple", "pear", "banana"]

Can have an initial size and value:

var numbs : [Int] = [Int] ( repeating: 5, count : 10 )

There are a number of useful properties and methods on arrays.

Properties: count, isEmpty

Methods: append, +=, insert, remove ( at ), removeLast

For example:

fruits.append( "mango" )

//OR

fruits += ["pineapple"]

fruits += ["strawberry", "raspberry"]

print( fruits )

print( fruits.count )

fruits.insert( "kiwi", at: 1 )

print( fruits )

fruits.removeLast()

fruits.remove( at : 1 )

print( fruits )

print( fruits.isEmpty )

You can set array values – as long as these elements already exist in the array.

var myArray:[Int] = [Int]( repeating: 4, count: 5 )

myArray[2] = 34

print( myArray )

myArray[2...3] = [21,22]

print( myArray )

myArray[1...1] = [1,2,3,4]

print( myArray )

For each loop in swift :

for f in fruits {

print( f )

}

Or you can count through using the index: (Why doesn’t work for String?)

for i in 0..<fruits.count

{

println( fruits[i] )

}

Given an array of grades, determine the average grade.

var grades = [ 80, 96, 55, 71, 42, 82, 93, 67 ]

var totalGrade = 0

for g in grades

{

totalGrade += g

}

var average = totalGrade / grades.count

println( "Average grade is \(average)" )