# LO5 Advanced Concepts

## Operator Overload Functions

In Swift == is used to check for equality of contents (like equals method in Java). If you want this operator to be available for your class you must overload the == operator.

Example:

class Pet

{ var name : String

var age : Int

init( name: String, age : Int )

{ self.name = name

self.age = age

}

}

func ==( left : Pet, right : Pet ) -> Bool

{ var result : Bool = false;

if left.name == right.name && left.age == right.age

{

result = true;

}

return result

}

var java = Pet( name : "Java", age : 8 )

var java1 = Pet( name : "Java", age : 8 )

print( java == java1 )

print( java === java1 )

var javaCopy = java

print( java == javaCopy )

print( java === javaCopy )

Being a Java programmer I find it very annoying that I can’t do this:

print( "Number is " + number )

But we can overload +!

func +( lhs : String, rhs : Int ) -> String

{

return "\(lhs)\(rhs)"

}

var number : Int = 22

print( "Number is " + number )

**Student Practice:** Write an operator overloaded function to do rounding when dividing two integers. Note: Function needs to appear before the code that uses it.

func /( lhs : Int, rhs : Int ) -> Int

{

var result : Double = Double( lhs) / Double(rhs )

result += 0.5

return Int( result )

}

var numb1 = 28

var numb2 = 10

print( numb1 / numb2 )

var dNumb1 : Double = 28

var dNumb2 : Double = 10

print( dNumb1 / dNumb2 )

## Tuples

Tuple is a simple way to treat a group of values as a single entity. Is a compound type, strongly typed but are not named. For example, we can store student name and grade in a tuple.

var student = ("Bob Brown", 75 ) //inferred types

var student1 : (String, Int )

student1 = ("Joe Jones", 61 )

print( student ) //Outputs complete tuple as ( “Bob Brown”, 75 )

print( student1 )

Can refer to individual elements as 0 and 1:

print( student.0 )

print( student.1 )

let name = student.0

let grade = student.1

Tuples can be nested, meaning a tuple may be made up of one or two other tuples. Tuples can be nested to any level but think about maintainability and understandability before going too far.

var appointment : ( String, ( Int, Int ) )

var appointment1 = ( "Friday", ( 2, 30 ) )

//Temporary variables can be used to access elements as well

let ( day, ( hour, minute) ) = appointment1

print( day )

print( hour )

print( minute )

Tuples can have any number of elements:

var match : ( String, Int, String, Int )

match = ("Winnipeg Jets", 4, "Anaheim Ducks", 0 )

print( match )

It can be useful to have labels for each part of the tuple – makes it less confusing:

var match : ( home: String, homeScore : Int, visitor : String, visitorScore : Int )

match = ("Winnipeg Jets", 4, "Anaheim Ducks", 0 )

print( match.visitor )

print( match.visitorScore )

//Numbers still work

print( match.0 )

## Tuples and Arrays

Can use a tuple with an array:

var students : [( String, Int)] = [(String, Int)] ()

students.append( ("John Smith", 91) )

students.append( ("Jane Johnson", 65) )

//But referring to name, grade in the following manner works?

var name = "Bobo Bobson"

var grade = 35

students[1...1] = [ ("Bob Brown", 85 ), ( "Sally Smith", 97 ), ( "Sue Smithison", 72 ), ( name, grade ) ]

print( students )

## Using Tuples with Switch statements

Tuples can be used in switch statements. If a value is not needed it can be ignored using \_. Cases do not need to be mutually exclusive – the first one that matches is used.

for s in students

{ switch( s )

{

case("Sally Smith", 58...100): print("Favourite, \(s.0) Student Passed!")

case( \_, 0...49 ) : print( "\(s.0) failed!")

case(\_, 50...59) : print("\(s.0) is in Supplemental range")

case(\_, 60...100 ) : print( "\(s.0) has passed")

default : print("Invalid Grade")

}

}

Example with tuple with more fields:

var match : ( String, Int, String, Int )

match = ("Winnipeg Jets", 5, "Anaheim Ducks", 4 )

print( match )

switch( match )

{

case let ( home, hScore, away, aScore) where hScore > aScore : print(" Home team, \(home), was victorious")

case let (home, hScore, visitor, aScore ) where aScore > hScore : print(" Visiting team, \(visitor), was victorious")

default : print( "Tie!")

}

**Student Practice:** Store name, number of cats and number of dogs in a tuple. Create an array with a few entries in it. Then loop through the array and using a switch statement display one of the following messages: “Name likes cats better than dogs” if they have more cats than dogs, “Name likes dogs better than cats” if they have more dogs than cats, “Name likes dogs and cats the same amount” if number of cats and dogs is the same AND greater than zero, “Name does not like cats or dogs” if number of cats and dogs is zero. If dogs and/or cats is zero display “invalid values”.

Solution:

var pets = [ ("Tom", 4, 5), ("Bob", 5,1), ("Sally", 1, 1), ("Jane", -1, 8), ("Darcy",0,0) ]

for elem in pets

{

switch( elem )

{

case let (name, cats, dogs) where cats < 0 || dogs < 0:

print( "Invalid values")

case let (name, cats, dogs) where cats > dogs:

print( "\(name) likes cats better than dogs")

case let (name, cats, dogs) where cats < dogs:

print( "\(name) likes dogs better than cats")

case let (name, cats, dogs) where cats == dogs && cats > 0:

print( "\(name) likes cats and dogs the same amount")

default:

print( "\(elem.0) does not like cats or dogs")

}

}

### Using Tuples with functions

You can use a tuple as a return value from a function allowing multiple pieces of data to be returned from the function. You can give the returned values names to easily refer to them after the method call. Although it is not necessary to give the returned values names – you can just refer to them as 0, 1, etc.

func mathFun( number : Int ) ->( sumDigits : Int, factorial : Int )

{

var facResult = 1

for i in 2...number

{

facResult = facResult \* i

}

var sumResult = 0

var numb = number

while( numb > 0 )

{

sumResult += numb % 10

numb /= 10 }

return ( sumResult, facResult )

}

var answer = mathFun( number : 5 )

print( answer.sumDigits )

print( answer.factorial )

print( answer.0 )

print( answer.1 )

answer = mathFun( number : 11 )

print( answer.sumDigits )

print( answer.factorial )

print( answer.0 )

print( answer.1 )

**Practice:**  Write a method that accepts the width and height and returns a tuple containing the area and perimeter. Use external names.

func calculateMeasurements(width : Int,height : Int ) -> (area : Int, perimenter : Int )

{

return ( width \* height, width \* 2 + height \* 2 )

}

# let result = calculateMeasurements( width : 10, height : 15 )

# print( result.area )

# print( result.perimenter )

# print( result )

# Extensions

Extensions are another way of adding extra functionality to a class without subclassing. Extensions allow new functionality to be added without access to the original source code. Extensions have no name. Extensions can add the following to a class or structure:

* Computed properties
* Methods
* Initializers
* Subscripts – allow you to use [] to refer to an element in class or structure.
* Nested types – define enum inside a struct for example so you could use an extension to include that enum in a struct

Any code that uses an extension must have access to the code that defines the extension. You can extend your own classes or classes that do not belong to you.

Note: Be careful with naming in extensions as name collisions can cause problems. Also, consider this when extending Swift classes – you may add a length method to String class but in a later release Apple may add a length method to their String class causing a collision with your method in your extension.

## Example – extend String to have a length method

extension String

{

public func length() -> Int

{

return self.characters.count

}

}

var name : String = "Sharon"

print( name.length() )

## Student Practice

Starting with the Room Struct previously created use an extension to add a perimeter property.

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)")

}

}

extension Room

{

public var perimeter : Int

{

return length \* 2 + width \* 2

}

}

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

print( bedroom.perimeter )

print( bedroom.area )

# Generics

Generics are supported just like in Java. One exceptions is that type MUST be specified – like when we used Array – we were forced to specify the data type stored within the Array. We can create our own classes that support Generics – then when our class is used we are required to specify the data type.

When defining a class that supports generics use a placeholder for any place where the data type is required. This placeholder is replaced with the appropriate data type when an object of this class is created.

For example: create a queue class that uses generics. Write first using Ints instead of E placeholders. Then show how to change.

class Queue<E>

{

var array : [E] = [E]()

var isEmpty : Bool { return array.isEmpty }//Computed Property

var numElements : Int { return array.count }//Computed Property

func enqueue( elem : E )

{

array.append( elem )

}

func dequeue( ) -> E

{

return array.remove( at: 0 )

}

}

var names = Queue<String>()

names.enqueue( elem: "Bob")

names.enqueue( elem: "Tom" )

names.enqueue( elem: "Jane")

print( names.isEmpty )

print( names.numElements )

print( names.dequeue() )

print( names.dequeue() )

print( names.dequeue() )

print( names.isEmpty )

print( names.numElements )

**Student Practice:**  Write a Stack class that uses generics. Have two computer properties – isEmpty and numElements. Have two methods, pop – remove last element add and push – add new element to top of stack.

class Stack<E>

{

var array : [E] = [E]()

var isEmpty : Bool { return array.isEmpty }

var numElements : Int { return array.count }

func push( elem : E ) { array.append( elem ) }

func pop( ) -> E

{

return array.remove( at : array.count - 1 )

}

}

var names = Stack<String>()

names.push( elem: "Bob")

names.push( elem: "Tom" )

names.push( elem: "Jane")

print( names.isEmpty )

print( names.numElements )

print( names.pop() )

print( names.pop() )

print( names.pop() )

print( names.isEmpty )

print( names.numElements )

## More on extensions: Example – adding subscript to Queue class

Note: Queue class in generics document

extension Queue

{

subscript(index: Int) -> E?

{

get

{

var result : E? = nil

if index >= 0 && index < numElements

{

result = array[index]

}

return result

}

set(newValue)

{

if index >= 0 && index < numElements

{

array[index] = newValue!

}

}

}

}

var lineUp : Queue<String> = Queue<String>()

lineUp.enqueue(elem : "Bob")

lineUp.enqueue( elem : "Tom")

lineUp[0] = "Joe"

print( lineUp[0] as Any )

print( lineUp[1] as Any )

**Student Practice:**  Write the MyString class that implements the CustomStringConvertible protocol. This class stores an Array of characters. It has two computed properties – length and description. Create two initializers – one takes in a String to initialize the character array and the other is just the default initializer which creates an empty character array. Add a subscript method for this class to access individual characters in the String – do both get and set for subscript, if an index is specified beyond the end of the existing String just use last character in String. Also, write a += function for this class to append on another String. See next page for solution.

# public class MyString : CustomStringConvertible

# {

# var characters : Array<Character> = Array<Character>()

# var length : Int { return characters.count }

# public var description: String { return String( characters ) }

# public init( inStr : String )

# {

# for c in inStr.characters

# {

# characters.append( c )

# }

# }

# public init(){//Do nothing: array initialized }

# subscript( index : Int ) -> Character?

# {

# get

# {//if index past end of String return last element

# var result : Character?

# result = (index >= characters.count )?nil:characters[index]

# return result

# }

# set(newValue)

# {

# if index < characters.count

# {

# characters[index] = newValue!

# }

# }

# }

# public func clear()

# {

# characters = Array<Character> ()

# }

# }

# func += ( left : inout MyString, right : String )

# {

# left.characters = Array<Character> ()//Clear character array

# for c in right.characters

# {

# left.characters.append( c )

# }

# }

# var name : MyString = MyString( inStr : "Sharon" )

# print( name[1] as Any )

# var pet : MyString = MyString()

# pet += "Java"

# print( pet )

# print( pet[0] as Any )

# Dictionary

A dictionary is a type of has table providing quick access to its entries. Key value pair.

var dict : [ String : Int ] //declare

dict = [:]//instantiate

dict["Sharon"] = 1234

dict["Shane"] = 1314

print( dict )

var test : Dictionary<String, Double> = Dictionary<String, Double>()

print( test )

print( dict.count )

## Search in Dictionary

print( dict["Sharon"] as Any ) //Returns optional value

print( dict["Tom"] as Any )

var dict2 : [ String : Int ] = [:]

dict2["Bob"] = 4004

dict2["Tom"] = 2121

for (key, value ) in dict2

//It would be nice to join this new dictionary to our original

{

dict.updateValue( value, forKey : key )

}

print( dict )

var dict3 : [String : Int ] = [:]

dict3["Sally"] = 45

dict3["Jane"] = 7099

extension Dictionary

//this might be what we want to do often add extension to Dictionary

{

mutating func append(other:Dictionary)

{

for (key,value) in other

{

self.updateValue(value, forKey:key)

}

}

}

dict.append( other: dict3 )

print( dict )

## Use tuples in Dictionary

You can use a tuple as the value in the pair allowing you to store information with a particular key.

var teams : [ String : ( Int, Int, Int ) ] = [:]//wins, losses, overtime losses

print( teams )

teams["Senators"] = ( 41, 24, 8 )

teams["Canadiens"] = (41, 24, 9 )

print( teams )

# Nested Functions

Functions can be nested within another functions. This is useful in some cases – particularly when you want to return a function type.

func changeByOne(backward: Bool) -> (Int) -> Int

{

func stepForward(input: Int) -> Int { return input + 1 }

func stepBackward(input: Int) -> Int { return input - 1 }

return backward ? stepBackward : stepForward

}

var currentValue = -4

let moveTowardZero = changeByOne(backward: currentValue > 0)

while currentValue != 0

{

print( currentValue )

currentValue = moveTowardZero(currentValue)

}

//Note nested functions are not available outside function

//currentValue = stepForward( currentValue ) //syntax error