Working with classes and objects

COSC346

ousa
otago uni students' association

SEMESTER TWO 2018

Are you

- proactive, friendly and keen to contribute to your learning environment?
- a great communicator who can represent your peers?

What's in it for you?

- Kudos & Karma
- Great friendships
- Access to FREE professional training opportunities and support
- A feed (or three)
- A reference letter from OUSA for your CV
- Invitations to Class Rep social events throughout the year

Don't wait any longer... sign up now!

Talk to your lecturer or email:

classrep@ousa.org.nz

Initialisation

- An object should be self-contained: independent and self-sufficient
  - Should allocate resources (memory) required for its operation
  - Should initialise its member variables to appropriate values
- Constructors
  - These are special methods invoked upon object creation
  - The place where the internal state of the object can be initialised
  - Typically these methods carry the same name as the class
  - Can take parameters, which allow user-defined initialisation

- Note: Destructors are methods that are automatically invoked when object is released from the memory—more about this when we discuss memory management.

Initialisation


```
class Complex {
    var real: Float
    var imag: Float

    init() {
        self.real = 0.0
        self.imag = 0.0
    }
}

var x: Complex
x = Complex()
```

Initialiser is invoked with the name of the class followed by initialiser arguments in parentheses

- In Swift constructors are referred to as **initialisers**
- Any class that uses stored properties must implement at least one initialiser
  - Compiler will give an error if a property is not initialised
- An object instance must be created through an initialiser
  - Compiler will not allow the use of an object that hasn't been initialised

Initialisation – multiple initialisers


```
class Complex {
    var real: Float
    var imag: Float

    init() {
        self.real = 0.0
        self.imag = 0.0
    }

    init(real: Float, imag: Float) {
        self.real = real
        self.imag = imag
    }

}

var x: Complex
x = Complex()
x = Complex(real: 1.0, imag: 2.3)
```


- There can be more than one initialiser
- The arguments in the initialisation call determine which initialiser is used
- The arguments (if there are any) must be named in the initialisation call

Initialisation – multiple initialisers


```
class Complex {
    var real: Float
    var imag: Float

    init() {
        self.real = 0.0
        self.imag = 0.0
    }

    init(real: Float, imag: Float = 0.0) {
        self.real = real
        self.imag = imag
    }

}

var x: Complex
x = Complex()
x = Complex(real: 1.0, imag: 2.3)
var y: Complex = Complex(real: -4.4)
```

- Initialiser arguments can be declared with a default value: these arguments can be omitted from initialisation call

Designated initialiser

- Often classes will have several available initialisation methods
- The **designated initialiser** is a method that is eventually invoked by all other initialisation methods
- Use of a designated initialiser lowers the chance of initialisation errors

Initialisation - multiple initialisers


```
class Complex {
    var real: Float
    var imag: Float

    convenience init() {
        self.init(real: 0.0)
    }

    init(real: Float, imag: Float = 0.0) {
        self.real = real
        self.imag = imag
    }

}

var x: Complex
x = Complex()
x = Complex(real: 1.0, imag: 2.3)
var y: Complex = Complex(real: -4.4)
```

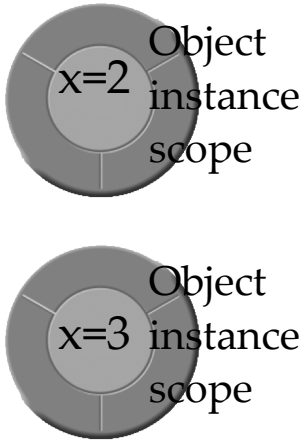
Convenience

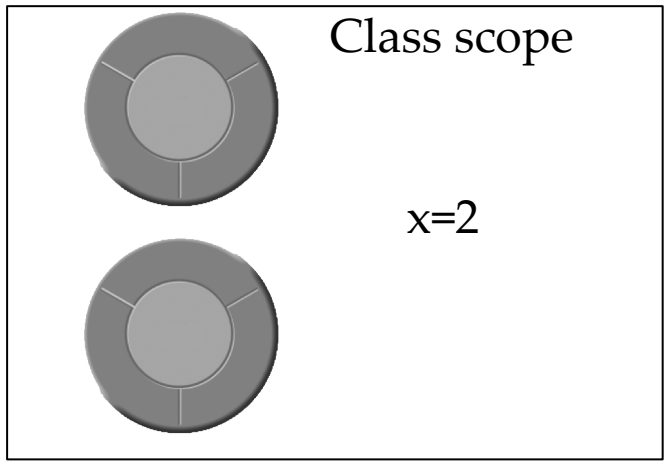
Designated

- In Swift an initialiser that doesn't write to properties directly, but initialises them through another initialiser is referred to as a convenience initialiser

Member/internal variables

- Member/internal variables are the variables encapsulated inside a class. There are two types of member variables:
  - Instance variables—each object instance carries a dedicated copy of these variables, and so values can different from object to object
  - Class variables—shared by all objects of a given class: change in value affects every instance





Member/Internal Variables

- Instance variables:
  - Stored properties—their values may differ for different object instances of the class
- Class variables:
  - Static stored properties—values are shared by all objects of a given class

```
class ClassA {
    static var firstTime: Bool = true; //Class variable
    var someProperty: Any              //Instance variable

    init(x: Any) {
        if(ClassA.firstTime) {
            //Do something that can be done
            //only once, regardless how many
            //objects of this class are created
            ClassA.firstTime = false
        }
        self.someProperty = x;
    }
}
```

Methods

- Instance methods
  - Methods invoked on class instance
  - Can access instance variables for read/write
- Class methods
  - Methods that can be used without creating an instance of the class
  - Cannot access instance variables

```
class Fraction {
  var num: Int;
  var den: Int;

  init(num : Int, den : Int) {
    self.num = num
    self.den = den
  }

  func add(_ f: Fraction) {
      self.num = self.num*f.den + self.den*f.num
      self.den = self.den*f.den
  }

  static func add(_ f1: Fraction, to f2: Fraction) -> Fraction {
        return Fraction (num: f1.num*f2.den + f1.den*f2.num,
                         den: f1.den*f2.den)
  }
}
```

```
var u = Fraction(num: 2, den: 3)
var v = Fraction(num: -7, den: 9)

u.add(v)

var f: Fraction
f = Fraction.add(u, to: v)
```

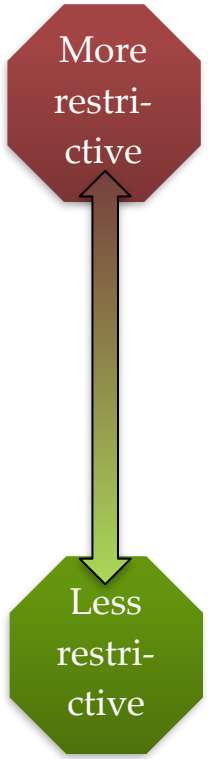
Operation on an instance

Operation on the class

Access Control

Generally in OOP access control is class based:

- **Private** methods and member variables
  - Accessible only from the code within class definition
  - Visible to the programmer writing the class code, not to the programmer using objects of that class
- **Protected** methods and member variables
  - A bit like private, except visible to derived classes—more about this in the lecture on inheritance
- **Public** methods and member variables
  - Accessible from anywhere
  - Visible to the programmer writing the class as well as the programmer using objects of that class

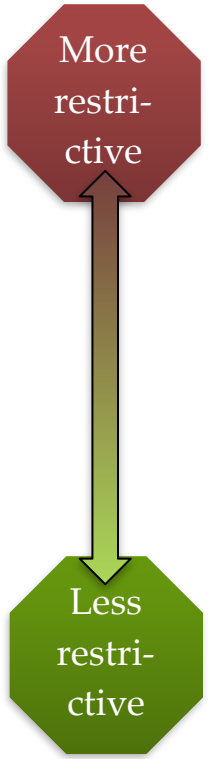


Access Control

Swift

In Swift access control is file/module based:

- **Private** methods and member variables
  - Accessible only from the file where the class has been defined
  - Visible to the programmer editing the class source file, even if working with object instances outside the class definition
- **Fileprivate** methods and member variables
  - Accessible only from the module where it's defined
- **Internal** methods and member variables (the **default**)
  - Accessible from any file that is part of the module where the class is defined where the class has been defined
  - Visible in any source file to the programmer writing within the module (project) where the class is defined
  - Once the module is distributed as a framework, and included in another project, the internal methods and variables are not accessible to the programmer using objects of that class
- **Open/Public** methods and member variables
  - Accessible from any source file
  - Visible to the programmer writing the class as well as the programmer using objects of that class in any source file of the project



Open vs Public

Open access applies only to classes and class members, and it differs from public access as follows:

- Classes with public access, or any more restrictive access level, can be subclassed only within the module where they're defined.
- Class members with public access, or any more restrictive access level, can be overridden by subclasses only within the module where they're defined.
- Open classes can be subclassed within the module where they're defined, and within any module that imports the module where they're defined.
- Open class members can be overridden by subclasses within the module where they're defined, and within any module that imports the module where they're defined.

https://docs.swift.org/swift-book/LanguageGuide/AccessControl.html

Accessor methods

- Sometime it makes sense to control the access to object's state
- The state of the object can be made private, so that the user of the object can't access it directly
- Setter & getter methods are the interface to the state of the object
  - These methods can check for indexes out of bounds, invalid values, etc., to ensure that state doesn't get corrupted

http://www.evinw.com

Accessor methods

```
class Fraction {
  private var _numHidden: Int = 0
  private var _denHidden: Int = 1

  init(num : Int, den : Int) {
    self._numHidden = num; self._denHidden = den
  }

  func getNum() -> Int {
    return _numHidden
  }

  func setNum(_ newValue: Int) {
    _numHidden = newValue
  }

  func getDen() -> Int {
    return _denHidden
  }

  func setDen(_ newValue: Int) {
    assert(newValue != 0, "Can't set den to 0!")
    _denHidden = newValue
  }
}
var f: Fraction = Fraction(num: 1, den: 3)
f.setNum(2)
f.setDen(3)
print("f=\(f.getNum())/\(f.getDen())")
```

Private state

Getter and setter for _numHidden

Getter and setter for _denHidden

- Generic setters and getters

Swift

Accessor methods

```
class Fraction {
  private var _numHidden: Int = 0;
  private var _denHidden: Int = 1;

  init(num : Int, den : Int) {
    self._numHidden = num; self._denHidden = den
  }

  var num: Int {
    get {
      return _numHidden
    }
    set(newValue) {
      _numHidden = newValue
    }
  }

  var den: Int {
    get {
      return self._denHidden
    }
    set(newValue) {
      assert(newValue != 0, "Can't set den to 0!")
      self._denHidden = newValue
    }
  }
}
var f: Fraction = Fraction(num: 1, den: 3)
f.num = 2
f.den = 3
print("f=\(f.num)/\(f.den)")
```

Private state

Getter and setter for _numHidden

Getter and setter for _denHidden

- Swift setters and getters as computed properties

Swift

Overloading

- Same method name but different implementations for different parameter signatures
- Constructor overloading is probably the most ubiquitous use of overloading

```
class Fraction {
    var num: Int
    var den: Int

    init(num: Int, den: Int) {
        self.num = num
        self.den = den
    }

    convenience init(string: String) {
        var num: Int = 0;
        var den: Int = 1;
        var tokens = string.components(separatedBy:"/")

        if tokens.count > 0 {
            if let n = Int(tokens[0]) {
                num = n
            }
        }

        if tokens.count > 1 {
            if let d = Int(tokens[1]) {
                den = d
            }
        }
        self.init(num: num, den: den)
    }
}

var x: Fraction = Fraction(num: 1, den: 2)
var y: Fraction = Fraction(string: "4/3")
```

Object reference

- What happens when you create an object instance?
- First, you create an object reference—if it's not initialised it doesn't point anywhere

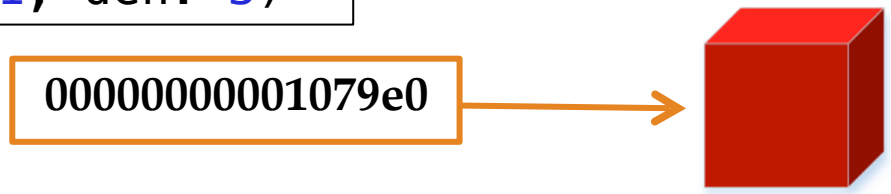
```
var f: Fraction
```

f

- Then, you create an object instance and set the reference value so it points to the address where the object is located

```
f = Fraction(num: 1, den: 3)
```

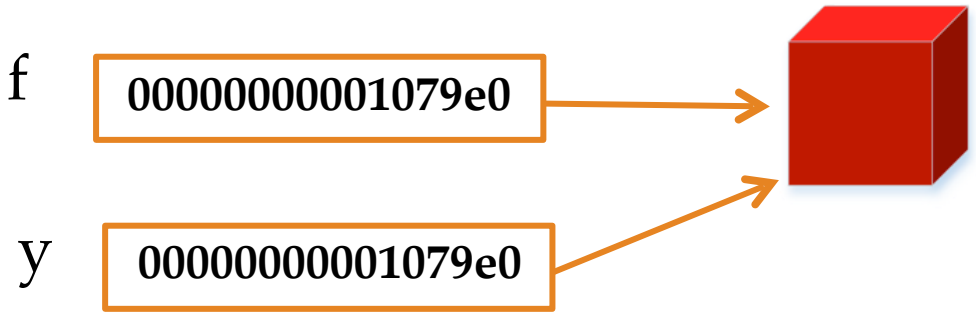
f



Object reference

- What happens when you assign an object reference to another reference?

```
var y: Fraction = f
```

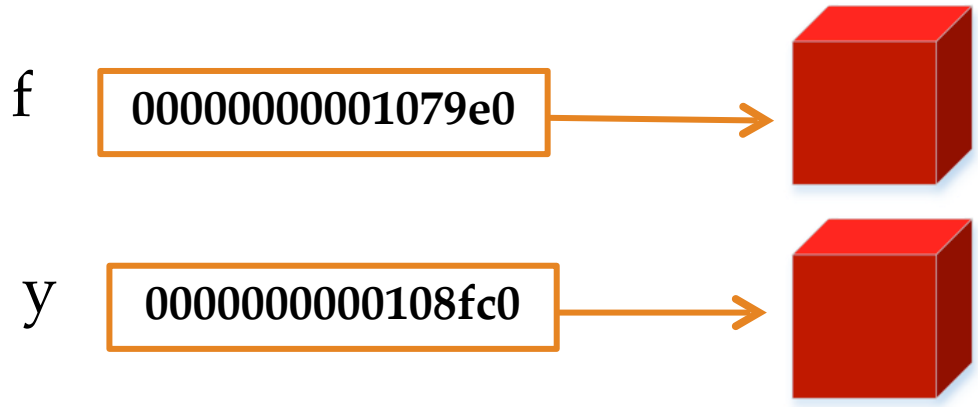


- Two references point to the same object instance: operation through either reference changes the state of the object

Copying objects

- Create a new object instance and copy the values of all instance variables

```
var y: Fraction = f.copy()
```



- Two references now point to different object instances: operation through one reference does not affect the object associated with the other reference

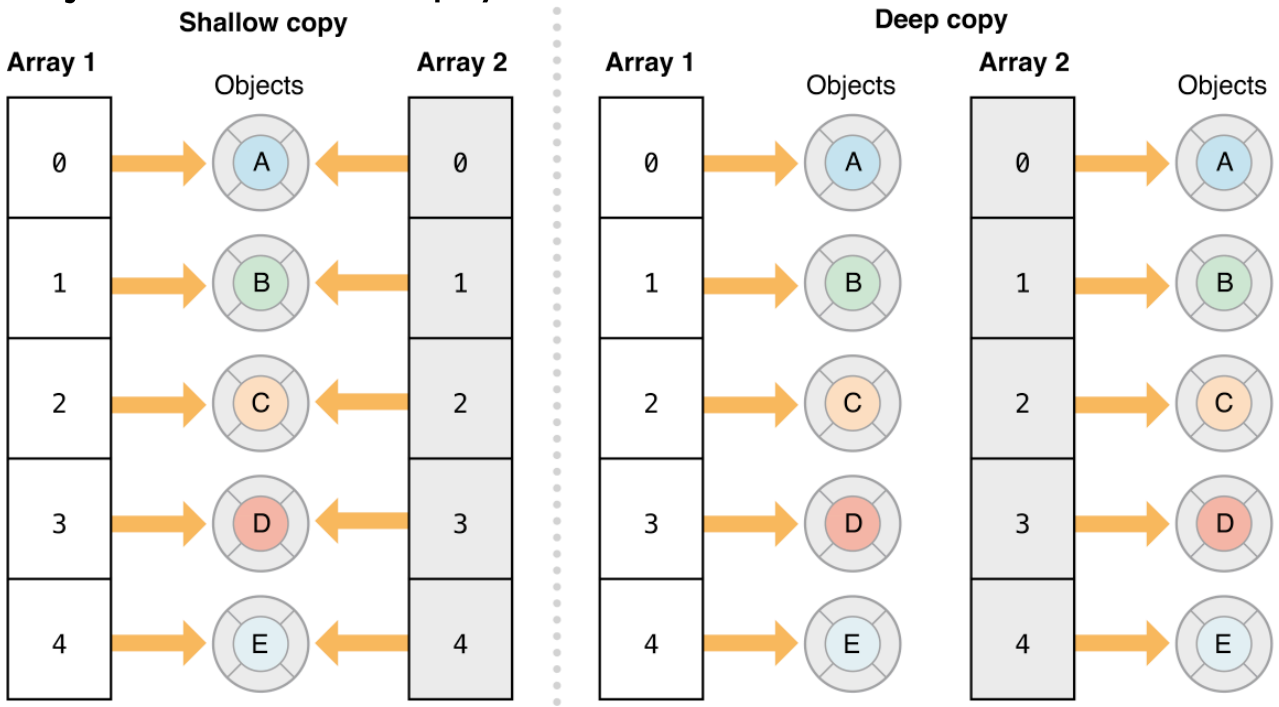
Copying objects

- Create a new object instance and copy the values of all instance variables

```
class Fraction {
  private var num: Int;
  private var den: Int;

  init(num : Int, den : Int) {
    self.num = num
    self.den = den
  }

  func copy() -> Fraction {
    return Fraction(num: self.num, den: self.den);
  }
}

var f: Fraction = Fraction(num: 1, den: 3)
var y = f.copy()
```

Copying objects

- What if an instance variable is an object?
  - Shallow copy—copy its reference
  - Deep copy—create new instance of the internal object and copy the state



Comparing object references

- Do you mean to check whether two references point to the same object instance?

```
var f: Fraction
var y: Fraction

.
.
.

if y === f {
    print("y and f refer to the same object")
}
```

Comparing objects' values

- What does it mean for different object instances to be equal?

```
func == (left: Fraction, right: Fraction) -> Bool {
    if (left.num == right.num) &&
        (left.den == right.den) {
            return true
    } else {
        return false
    }
}

var f: Fraction
var y: Fraction

.
.
.

if y == f {
    print("Objects are the same")
}
```

Comparing objects by value

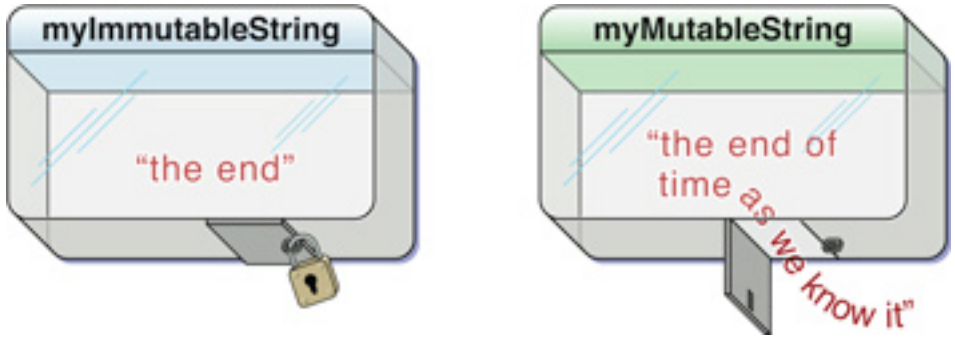
- Can the objects be ordered in some fashion?

```
func < (left: Fraction, right: Fraction) -> Bool {
    if left.decimal < right.decimal {
        return true
    } else {
        return false
    }
}

var f: Fraction
var y: Fraction

.
.
.

if y < f {
    print("y is smaller than f")
} else {
    print("f is smaller than y")
}
```

Computed property that calculates a float value from Fraction's state

Mutable and immutable

- Mutable object—internal state can be modified at any point
- Immutable object—internal state does not change after initialisation: a read only object



Mutable and immutable

- Declaring stored properties as constant makes the class objects immutable
  - Properties can only be set once in an initialiser

```
class Fraction {
    let num: Int;
    let den: Int;

    init(num : Int, den : Int) {
        self.num = num
        self.den = den
    }

    func copy() -> Fraction {
        return Fraction(num: self.num, den: self.den);
    }

    static func add(f1: Fraction, to f2: Fraction) -> Fraction {
        return Fraction (num: f1.num*f2.den + f1.den*f2.num,
            den: f1.den*f2.den)
    }
}

var f: Fraction = Fraction(num: 1, den: 3)
var y = f.copy()
y.num = 2
print ("f=\(f.num)/\(f.den)")
```

Stored properties declared as constants

The only place where the stored properties can be set

Cannot assign to 'num' in 'y'

Pattern of the Day - Singleton

- **Creational**
  - Only allow creation of a limited number of instances of a class (usually just one)

- Often abused/misused
- Considered to be an *anti-pattern*

What OOP principle does the singleton break?

```java
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

```java
public final class Singleton {
    private static Singleton instance = null;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Swift

```swift
class DeathStarSuperlaser {

    static let sharedInstance = DeathStarSuperlaser()

    private init() {
        // Private initialization to ensure just one
        // instance is created.
    }
}

let laser = DeathStarSuperlaser.sharedInstance
```

https://github.com/ochococo/Design-Patterns-In-Swift/blob/master/source/creational/singleton.swift

Summary?

Summary

- Instantiation
- Access Control
- Overloading
- Object References and Copying
  - Deep vs Shallow
- Immutability
- Design Pattern - Singleton