Object Oriented Programming Review

COSC346

PARTICIPANTS NEEDED FOR COURSE ADVICE STUDY

The project explores the factors influencing undergraduate students' decisions regarding their path of study, e.g., who guides their decisions, how confident are they with these decisions, and how can the course advising process be improved to better suit their needs.

The project entails an initial short profiling questionnaire (5 mins), followed by a focus group session (up to 1 hour long).

You will be given the following reimbursement:

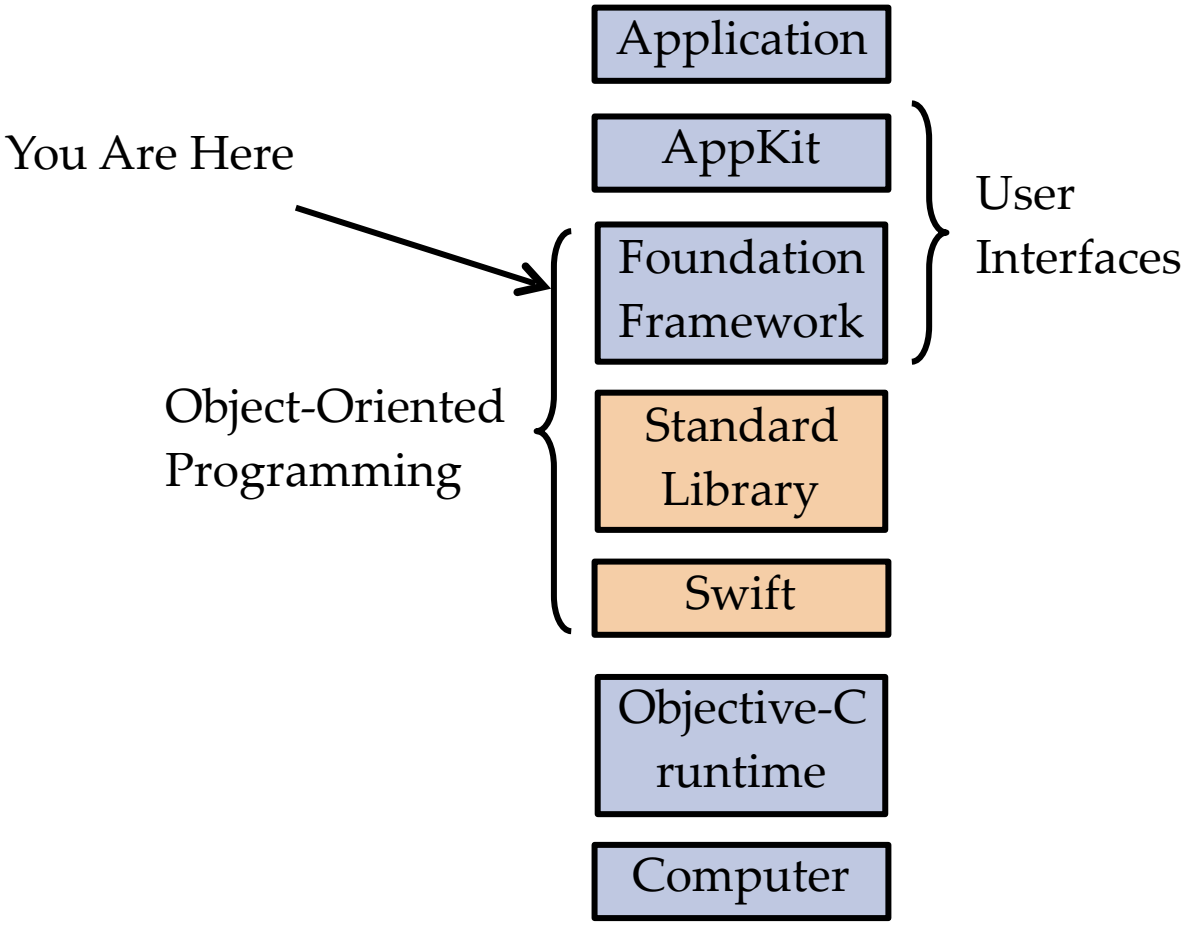
We are seeking 15-20 participants for this project, which is entitled 'Data driven course advising'. If you are:

- On-campus full time undergraduate information science and computer science students,
- Students either in the first year of their degree or in their final year.

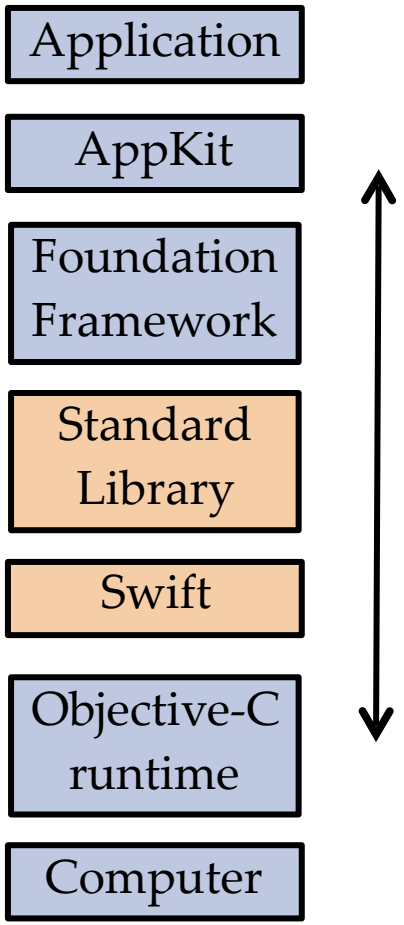
$20 New World voucher!

If you interested in participating please email: Senorita John: senorita.john@postgrad.otago.ac.nz by Friday (24th August) this week.

OOP Review


Application
AppKit
Foundation Framework
Standard Library
Swift
Objective-C runtime
Computer
You Are Here
User Interfaces
Object-Oriented Programming

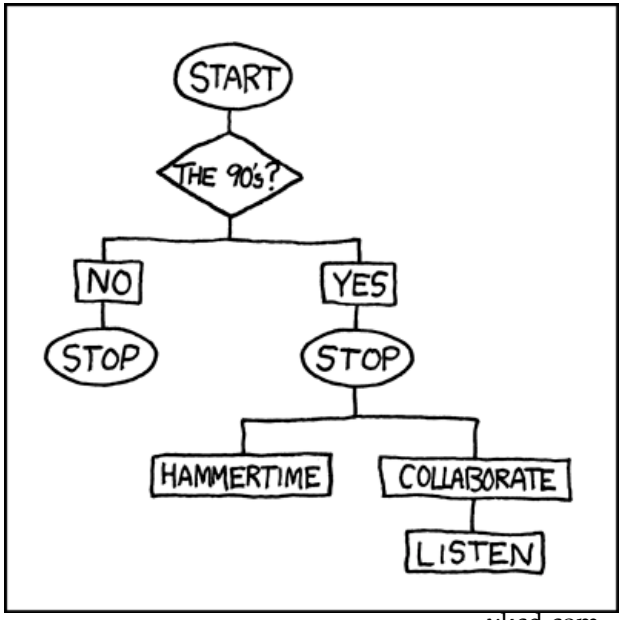
Why OOP



Speed	Code	Development	Environment	User Interface
Slow	Re-usable	Team, Fast	Runtime Decisions	Complex, Graphical
Fast	Specific	Individual, Slow	Compile-time Decisions	Simple, Text-based

Procedural versus Object-Oriented

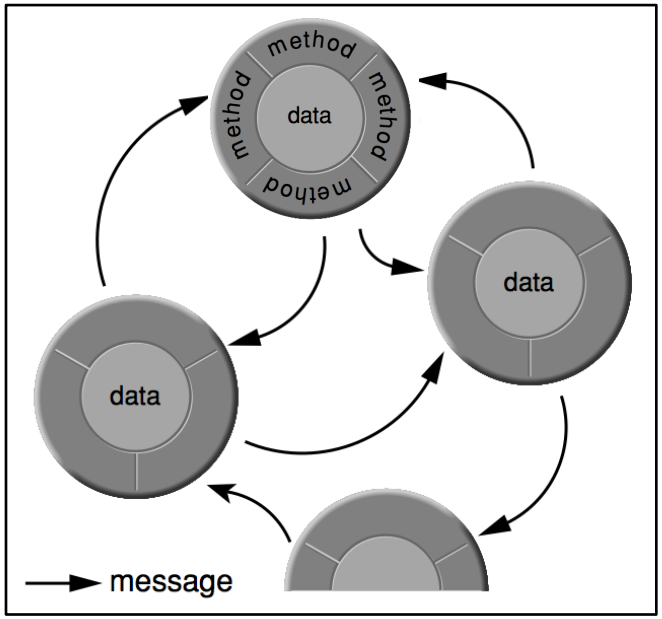
Procedural



xkcd.com

1. Functions act on data
2. A program organises function calls to manipulate data

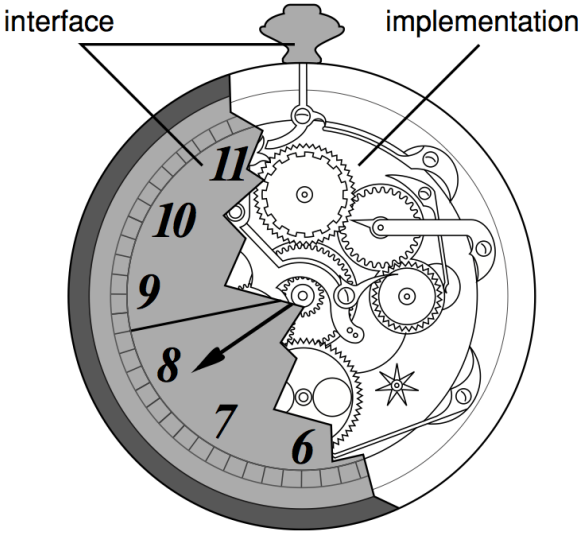
Object-Oriented



1. *Objects* contain *encapsulated* data and associated *methods*
2. A program describes how objects interact via *messages*

Classes and Objects

- Abstraction
  - What the object does is more important than how it does it
- Encapsulation
  - Control how the internal state of the object is accessed
  - Accessor methods
    - setters
    - getters
  - Visibility
    - public, private, (fileprivate,) protected, internal variables & methods
- Interface versus implementation



Working with objects

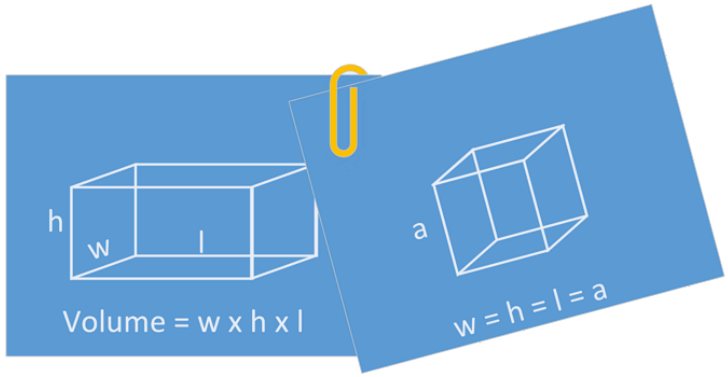
- Constructors
  - Create and initialise an object instance
- Destructors
  - Clean up after object is decommissioned
- References
  - Pointer to object's location in memory
  - There can be many references to the same object
- Copying objects
  - Simple assignment might just copy object reference
  - Need a method that copies the internal state
    - Shallow copy
    - Deep copy

Working with objects

- Comparing objects
  - Comparing references only establishes if they point to the same object instance
  - Need a method that captures the meaning of relative order based on the internal state
- Mutability
  - The state of a mutable object may change
  - The state of an immutable object cannot change (aside from at initialisation time)
  - Immutability is somewhat analogous to constants
- Serialisation
  - Process of converting an object into a data stream

Inheritance

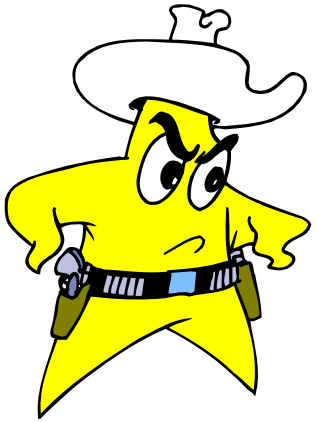
- Good reasons for using inheritance
  - Specialisation, specification, extension
- Not so good reasons for using inheritance
  - Limitation, generalisation
- Costs and benefits of using inheritance
- Multiple inheritance
  - Diamond of death
- Inheritance vs. composition
  - Is-a vs. has-a relationship
- Upcasting and downcasting



Memory management

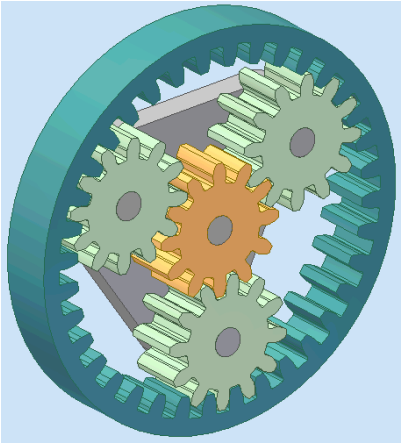
- Stack versus heap
  - Local memory versus global memory
- Object ownership
  - Who is responsible for destroying the object?
- Reference counting
  - Keep track of how many references there are in a program to an object—destroy the object when no one is referencing it
  - Retain cycles break reference counting
  - Thus weak and strong references introduced
- Contrast approaches: Manual / Garbage collector / Automatic Reference Counting

Polymorphism



- Different classes, same methods
- Overloading
  - Same function name, different arguments
- Overriding
  - Same function name, same arguments, different implementation in a different subclass
- Generics
- Protocols / Interfaces / Abstract classes
- Introspection / Reflection
  - Queries about object type, methods, *etc*.

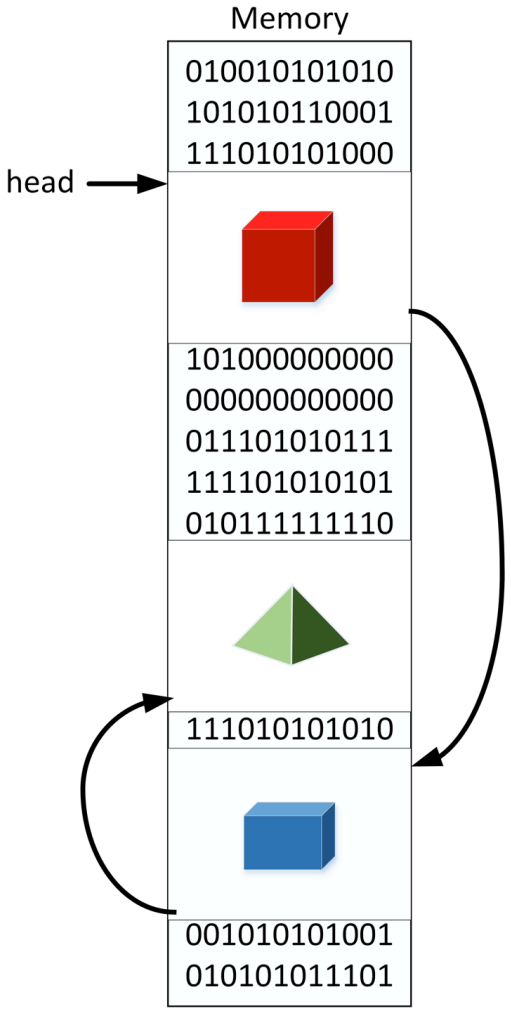
Object interconnection



- Cohesion
  - How well the object internals go together
- Coupling
  - The degree of inter-dependency between objects
- Design goals: loose coupling and high cohesion
- Callbacks
  - Arguments to methods that reference executable code
- Delegates
  - References to objects that implement a number of methods to use as callbacks

Standard libraries

- Collections/containers
  - Classes that hold objects
- Common containers
  - Lists
  - Arrays
  - Queues
  - Stacks
  - Maps/Dictionaries
  - Sets
- Serialisation



Object-oriented design

- What are use-cases?
- Sequence diagrams
- Class diagrams

Object-oriented design patterns

- Design patterns versus algorithms
- Behavioural/Creational/Structural

1. Iterator
2. Singleton
3. Strategy
4. Façade
5. Factory
6. Flyweight
7. Observer
8. Command
9. Memento
10. Decorator
11. MVC
12. Chain of Responsibility

Design Pattern - Chain of Responsibility

- Process varied requests
- May be dealt with by different handlers

- Click event dispatch in GUI applications
  - start at child
  - bubble up to parents
  - can stop propagation at any point

- Different to decorators!!!

Chain of Responsibility Example

Toolmaker

```swift
final class MoneyPile {

    let value: Int
    var quantity: Int
    var nextPile: MoneyPile?
    init(value: Int, quantity: Int, nextPile: MoneyPile?) {
        self.value = value
        self.quantity = quantity
        self.nextPile = nextPile
    }

    func canWithdraw(amount: Int) -> Bool {
        var amount = amount
        func canTakeSomeBill(want: Int) -> Bool {
            return (want / self.value) > 0
        }
        var quantity = self.quantity
        while canTakeSomeBill(want: amount) {
            if quantity == 0 {
                break
            }
            amount -= self.value
            quantity -= 1
        }

        guard amount > 0 else {
            return true
        }
        if let next = self.nextPile {
            return next.canWithdraw(amount: amount)
        }
        return false
    }
}
```

```swift
final class ATM {
    private var hundred: MoneyPile
    private var fifty: MoneyPile
    private var twenty: MoneyPile
    private var ten: MoneyPile

    private var startPile: MoneyPile {
        return self.hundred
    }

    init(hundred: MoneyPile,
         fifty: MoneyPile,
         twenty: MoneyPile,
         ten: MoneyPile) {

        self.hundred = hundred
        self.fifty = fifty
        self.twenty = twenty
        self.ten = ten
    }

    func canWithdraw(amount: Int) -> String {
        return "Can withdraw:
            \(self.startPile.canWithdraw(
                amount: amount))"
    }
}
```

Builder

```swift
let ten = MoneyPile(10, 6, nil)
let twenty = MoneyPile(20, 2, ten)
let fifty = MoneyPile(50, 2, twenty)
let hundred = MoneyPile(100, 1, fifty)
```

```swift
var atm = ATM(hundred, fifty, twenty, ten)
print(atm.canWithdraw(amount: 310)) // false
print(atm.canWithdraw(amount: 100)) // true
print(atm.canWithdraw(amount: 165)) // false
print(atm.canWithdraw(amount: 30))  // true
```

CoR in the real world?