Object interconnections

COSC346

Coupling

- **Coupling** describes how much different components of a system, such as objects in a program, depend on each other
  - For example, the relationship between classes

- Examples of coupling include:
  - **Internal data coupling**—direct access from one class to an instance variable of another class
  - **Global data coupling**—dependency on global variables
  - **Sequence coupling**—order of operation is vital, but not implicit in the class implementation

Cohesion

- Cohesion describes how well components, such as instance variables of the class, belong together
  - *e.g.*, the relationship between internal elements of a class

- Examples of cohesion:
  - **Coincidental cohesion**—elements of a class are grouped for no apparent reason
  - **Logical cohesion**—logical connection, no actual connection between data nor control
  - **Temporal cohesion**—elements that must be used at approximately the same time
  - **Data cohesion**—many elements serve to implement one data abstraction

Object Interconnection

- Coupling and cohesion relate to the way you design your class infrastructure
  - Want to minimise coupling and maximise cohesion
- **Loose coupling**:
  - Objects don't need to know much about each other to interact
  - Changes in one class are less likely to affect other classes
- **High cohesion**:
  - Easier to use classes that have single, clear purpose
  - Well modularised code is straightforward to maintain

Visibility

- Visibility has to do with what data/methods are accessible/available to the class user and what is hidden
- You can control visibility by declaring instance variables and/or methods as:
  - **Public**—class user has access
  - **Internal**—only accessible within same module
  - **Private**—only accessible from within same file
- Reducing visibility tends to reduce coupling

Visibility

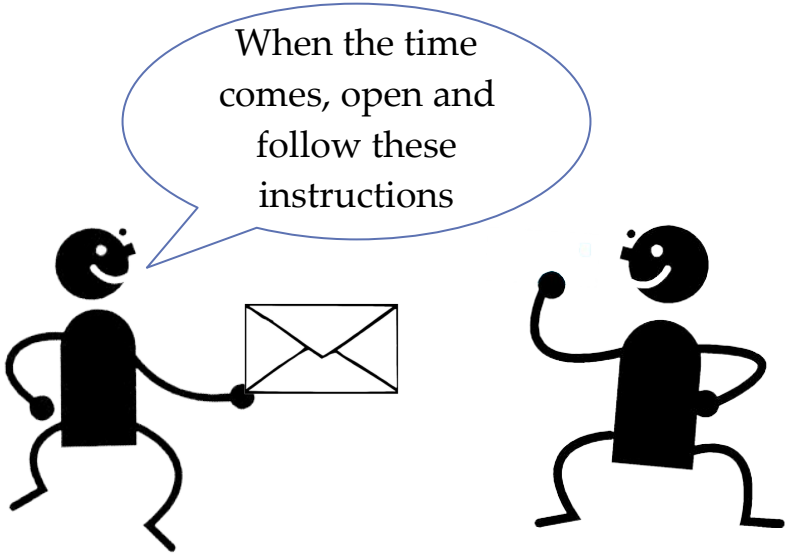
Swift

- In Swift functions, classes, methods, and properties can be made:
  - open
  - public
  - internal
  - fileprivate
  - private

```
public class GameObject {
  // Private
  private var _renderActions: [SKAction] = []
  private var _collider: Collider?
  // Internal
  var scene: Scene?
  var parent: GameObject?
  var children: [GameObject] = []
  var render: SKNode?
  var destroyMe: Bool = false
  // Public
  public var position: CGPoint = CGPoint(x: 0, y: 0)
  public var collider: Collider? {
        get {
            return self._collider
        }
        set(newCollider) {
            if let collider = newCollider {
                self._collider = collider
                self.addChild(collider)
            }
        }
  }
}
```

Callback

- A **callback** is a function that is passed as an argument to another function/method
- Typically callbacks are functions with instructions for what needs to happen after an asynchronous event takes place



- You can think of callbacks as plugins that extend the functionality of the caller object

Callback Example


```
// these lines are needed so the playground keeps running to let the
// callbacks complete
import PlaygroundSupport
PlaygroundPage.current.needsIndefiniteExecution = true

// this is some operation that can take a long time. Something like Network
// IO. I've simulated it with a call to 'sleep'
func someReallyLongOperation(key: String){
    print("SRLO: \(key) - starting a long operation")
    sleep(10)
    print("SRLO: \(key) - finished a long operation")
}

print("starting operation")
someReallyLongOperation(key:"synchronous with no callback")
print("... back")
```

starting operation
SRLO: synchronous with no callback - starting a long operation
SRLO: synchronous with no callback - finished a long operation
... back

Callback Example

Swift

```
// depending on the design of the API, you may have to do this yourself
func someReallyLongOperationWithCallbacks(onDone: () -> Void, key: String){
    someReallyLongOperation(key:key)
    onDone()
}

func callback(){
    print("CB: in the callback")
}

print("starting operation")
someReallyLongOperationWithCallbacks(onDone: callback, key: "synchronous with callback")
print("... back")
```

starting operation
SRLO: synchronous with callback - starting a long operation
SRLO: synchronous with callback - finished a long operation
CB: in the callback
... back

Callback Example

Swift

```
print("starting dispatch queue")
DispatchQueue.global(qos: .default).async {
    someReallyLongOperationWithCallbacks(
      onDone: callback,
      key: "asynchronous with callback"
    )
}
print("... back")
```

starting dispatch queue
... back
SRLO: asynchronous with callback - starting a long operation
SRLO: asynchronous with callback - finished a long operation
CB: in the callback

Callback Example

Swift

```
print("starting dispatch queue")
DispatchQueue.global(qos: .default).async {
    someReallyLongOperationWithCallbacks(
      onDone: {
        print("this is another callback")
      },
      key: "asynchronous with callback")
}
print("... back")
```

starting dispatch queue
... back
SRLO: asynchronous with callback - starting a long operation
SRLO: asynchronous with callback - finished a long operation
this is another callback

Callback Example - Real World

Swift

- The `URLSession` class implements a `dataTask` method that returns a task handler, which can be 'resumed' to initiate a URL data request
- The `completionHandler` callback is executed when the data is received

```
let url = URL(string: "http://cs.otago.ac.nz")

func myCallback(data: Data?, response: URLResponse?, error: Error?) {
    if let rcvdData = data {
        if let dataStr = NSString(data: rcvdData as Data, encoding:
                                  String.Encoding.utf8.rawValue) {
            print(dataStr)
        }
    }
}

var session = URLSession.shared
var task = session.dataTask(with: url!, completionHandler: myCallback)
task.resume()
```

Selector Callback

Swift

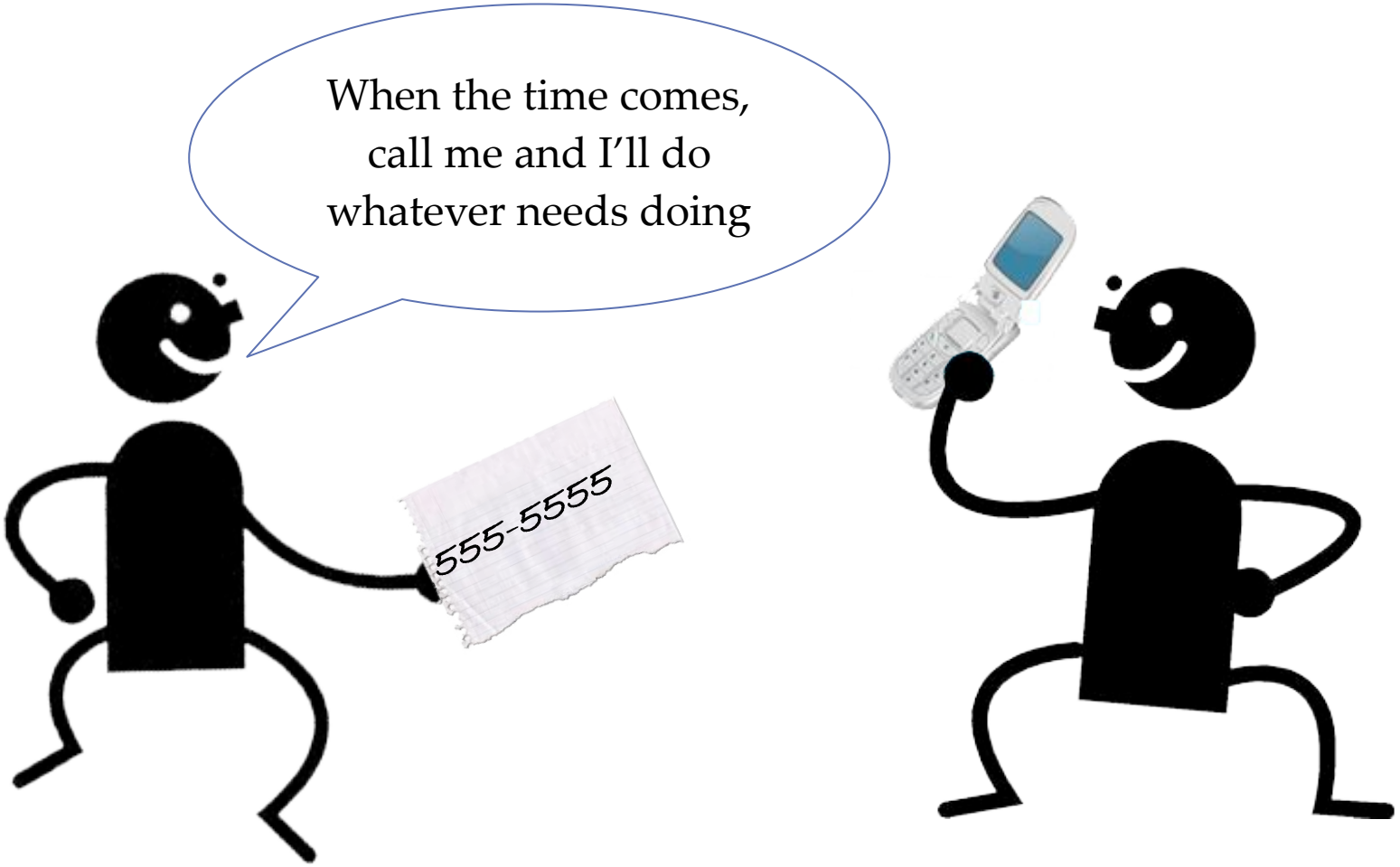
- Methods are also referred to as selectors
- `Thread` can invoke a selector of a target object and execute it in a new thread
  - The object that implements the selector callback must inherit from `NSObject`

```
class Worker: NSObject {

    func doWork() {
        while true {
            print("Work, work!")
            Thread.sleep(forTimeInterval: 0.2)
        }
    }
}

let peon = Worker ()
let thread = Thread(target:peon,
                    selector:#selector(Worker.doWork),
                    object:nil)
thread.start()
```

Delegation

- Another form of callback in Swift is delegation
- A **delegate** is an object implementing a number of callback methods
  - Instead of registering each callback method, you register the delegate object
  - Given the delegate's reference, the caller object can invoke various methods on the delegate object
- "Delegate acts on behalf of the caller object"
- Protocols are useful for delegation: you can specify the methods that the delegate must implement
- The Cocoa Framework relies heavily on delegation

Delegation


When the time comes, call me and I'll do whatever needs doing
555-5555

Delegation Example

- `URLSession` class can be instantiated with a delegate that provides methods that handle various events associated with the session
- Delegate must conform to the `URLSessionDataDelegate` protocol
- Delegate's `urlSession(session: URLSession, dataTask: URLSessionDataTask, didReceiveData: Data)` method is invoked when URL request returns data

```
let url = URL(string: "http://cs.otago.ac.nz")

public class MyDelegate: NSObject, URLSessionDataDelegate {

    public func urlSession(_ session: URLSession,
                           dataTask: URLSessionDataTask,
                           didReceive data: Data) {
        if let dataStr = NSString(data: data, encoding: String.Encoding.utf8.rawValue) {
            print(dataStr)
        }
    }
}

let delegateObj = MyDelegate()

var session = URLSession(configuration: URLSessionConfiguration.default,
                           delegate: delegateObj,
                           delegateQueue: nil)
var task = session.dataTask(with: url!)
task.resume()
```

Design Pattern - Observer

- Behavioural pattern
- Publish changes to object's state
- Subscribers receive notifications

- Two parts: Observer, Observable

Design Pattern - Observer

- Behavioural pattern
- Publish changes to object's state
- Subscribers receive notifications

- Two parts: Observer, Observable

```
protocol Observer{
    func onChange()
}
```

```
protocol Observable{
    func addObserver(o: Observer)
    func removeObserver(o: Observer)
}
```

Property Observers


```
class Person{
    var name:String {
        willSet{
            print("name will be changed from \(name) to \(newValue)")
        }
        didSet {
            print("name was changed from \(oldValue) to \(name)")
        }
    }
    init(name: String){
        self.name = name
    }
}

var person = Person(name: "John")
person.name = "Jack"
```

```
name will be changed from John to Jack
name was changed from John to Jack
```

Observer Example

Toolmaker

```
protocol Observer{
    func onChange(old: Any, new: Any)
}

class Person {
    var observers: [Observer] = []
    var name:String {
        didSet {
            for o in observers {
                o.onChange(old: oldValue, new: name)
            }
        }
    }
    // skipping init
    func addObserver(obs: Observer){
        self.observers.append(obs)
    }
}
```

Builder

```
class PersonNameChangeObserver: Observer{
    func onChange(old: Any, new: Any) {
        print("\(old) -> \(new)")
    }
}
```

```
var person = Person(name: "John")
var o = PersonNameChangeObserver()
person.addObserver(obs: o)
person.name = "Jack"
```

Observer in the real world?

Summary?