Polymorphism

COSC346

Polymorphism

- **Polymorphism** refers to the ability of different class objects to respond to the same method(s)
  - From the perspective of the message sender, the receiver can take different forms, as long as it implements the same methods
  - These methods may operate in different ways, but provide analogous behaviour

- Often used to provide similar functionality for different objects—the internal behaviour might differ, but the external interface (and, to certain extent, functionality) is the same

Polymorphism

Swift

Swift example: **description** method

- by convention this is a computed property returning a string meant to describe object contents
- for some objects this might just be the object's address, for others, a description of the object state
- The statement:

```
func show(x: AnyObject) {
    print(x.description);
}
```

- is meant to perform the same function—show a string representation of the object—regardless of the object type

Polymorphism

- **Overloading**—same method name, different implementations for different signatures
- **Overriding**—same method name, same signature, different implementation for different position in inheritance hierarchy
- **Upcasting**—same object, different types as long as its a parent type from inheritance hierarchy

- Polymorphic variable
- Generics
- Protocols

Polymorphic variable

- Can hold values of different types (type depends on the context)
- Examples in Swift:
    - `self`
    - `super`

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

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

`self` refers to an object of type `Complex`

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

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

`self` refers to an object of type `Fraction`

Dynamic versus static typing

- **Static typing**—data type derived from variable definition
  - Compiler checks for type mismatches spotting potential bugs
  - Type must be always specified
  - Cannot compile the code with a type mismatch
- **Dynamic typing**—data type is derived from its value
  - Type checking is deferred until run-time
  - Allows generic code that works with any type
  - Potential bugs might lurk in the code and not manifest until specific run-time conditions
- C++ and Java are statically typed languages
  - Objective C is dynamically type-checked, but allows programmer to enforce static type-checking
  - **Swift** is **statically typed** but "**with a dynamic feel**": type can be implicit wherever compiler can infer it

Static typing with dynamic flavour

Swift

x is an 'Int'

```
var x = 3
x += 2
x += "2"      Binary operator '+=' cannot be applied to operands of type 'Int' and 'String'
x = "3"       Cannot assign a value of type 'String' to type 'Int'
```

x is an 'Int' type cast to 'Any'

```
var x: Any = 3
x += 2        Binary operator '+=' cannot be applied to operands of type 'Any' and 'Int'
x += "2"      Binary operator '+=' cannot be applied to operands of type 'Any' and 'String'
x = "3"
```

x is a 'String' type cast to 'Any'

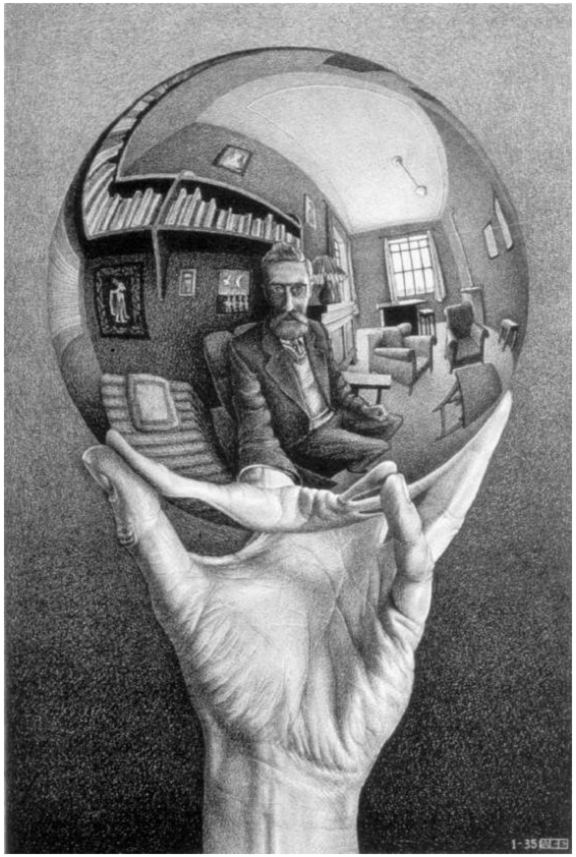
Reflection/Introspection

- Sometimes there is a need to ask objects about themselves at run-time

```
func += (left: inout Any, right: Int) {
    if left is Int {
        let leftInt = left as! Int
        left = leftInt + right;
    } else if left is String {
        let leftString = left as! String
        left = leftString + "\(right)"
    }
}

var x: Any = 3
x += 2
print("x=\(x)")

var y: Any = "3"
y += 2
print ("y=\(y)")
```



Reflection/Introspection

Swift

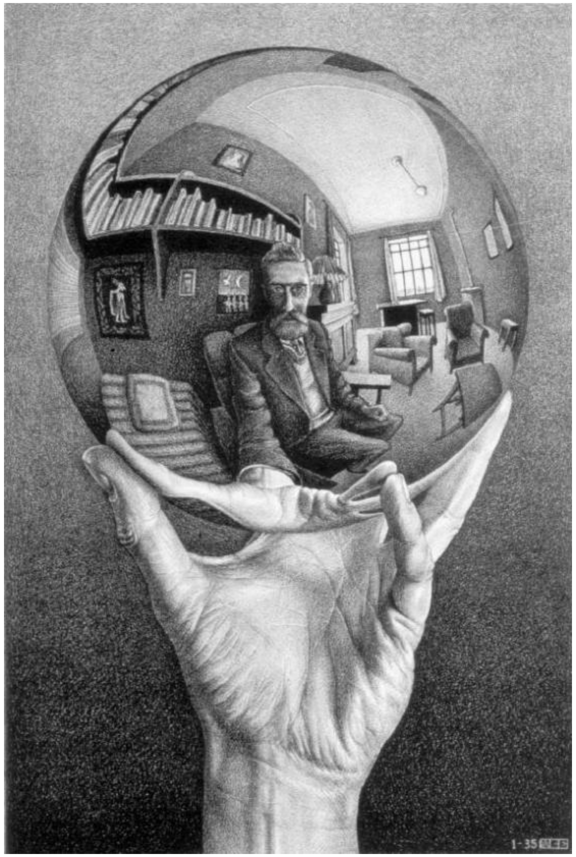
- Can inherit from NSObject, which provides more methods for introspection

```
class Fraction: NSObject {
  private let num: Int;    // Numerator
  private let den: Int;    // Denominator
  init(num : Int, den : Int) {
    self.num = num
    self.den = den
  }

  func add(f: Fraction) -> Fraction {
    return Fraction(num: self.num*f.den+self.den*f.num,
                    den: self.den*f.den)
    }
}
```

```
let f = Fraction(num: 2, den: 5)

if f.responds(to: #selector(Fraction.add(f:))){
    print("Responds to add")
} else {
    print("Does not respond to add")
}
```



Generics

- Also referred to as **parametric polymorphism**
- Generics provide ability to operate on generic data types
- Class definition includes a generic type, which allows one to create a library with unspecified data type
- The code that uses the library specifies the desired data type in place of the generic one
- References to the generic type inside the library code become references to the specified data
- Compiler checks for type consistency

Generics

Swift

Definition of generic type

```
class SimpleDictionary<T> {

    typealias Entry = (key: Int, item: T)

    var data: [Entry] = []

    subscript(key: Int) -> T? {
      get {
        for entry in data {
          if entry.key == key {
            return entry.item
          }
        }
        return nil
      }
      set(newItem) {
        if let item = newItem {
          for i in 0..<data.count {
            if data[i].key == key {
              data[i].item = item
              return
            }
          }
          data.append(Entry(key: key, item: item))
        }
      }
    }
}
```

Usage of generic type

Dictionary of Strings

Dictionary of Fractions

```
import Foundation

var dict1 = SimpleDictionary<String>()
var dict2 = SimpleDictionary<Fraction>()

dict1[2] = "item X"
dict1[435] = "item Y"

if let item = dict1[435] {
    print(item)
}

dict2[97] = Fraction(num: 1, den: 2)
dict2[21] = Fraction(num: 1, den: 3)

if let item = dict2[21] {
    print(item)
}
```

Protocols

Swift

- Protocols are a feature of Swift which allow enforcement of polymorphic behaviour
- A protocol definition lists a group of mandatory and optional methods

```
protocol Equatable {
  func ==(lhs: Self, rhs: Self) -> Bool
}

protocol Hashable : Equatable {
  var hashValue: Int { get }
}
```

- In class definition you can specify the protocols that the class will conform to

Will conform to these protocols

```
class Fraction: Hashable, CustomStringConvertible {
```

Protocols

Swift

```
class Fraction: Hashable, CustomStringConvertible {

    private let num: Int;    // Numerator
    private let den: Int;    // Denominator

    var decimal: Float {
        return Float(num)/Float(den)
    }

    var description: String {
        return "\(self.num)/\(self.den)"
    }

    var hashValue: Int {
        return num*den+den
    }

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

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

Required by `CustomStringConvertible`

Required by `Hashable`

- The class must implement the methods specified in the protocol it subscribes to, otherwise the compiler will complain

Generics with Protocol constraints

Swift

- Generic types can be constrained so that they must conform to desired protocol(s)

```
import Foundation

class SimpleDictionary<U: Hashable, T> {

  typealias Entry = (key: U, item: T)

  var data: [Entry] = []

  subscript(keyObject: U) -> T? {
    get {
      for entry in data {
        if entry.key == keyObject {
           return entry.item
        }
      }
      return nil
    }
    set(newItem) {
      if let item = newItem {
        for i in 0..<data.count {
          if data[i].key == keyObject {
            data[i].item = item
            return
          }
        }
        data.append(Entry(key: keyObject, item: item))
      }
    }
  }
}
```

Will work with types that conform to the Hashable protocol

```
var dict1 = SimpleDictionary<Fraction, String>()

var k1 = Fraction(num: 1, den: 3)
var k2 = Fraction(num: 2, den: 5)

dict1[k1] = "item X"
dict1[k2] = "item Y"

if let item = dict1[k1] {
    print(item)
}
```

Class must conform to Hashable protocol

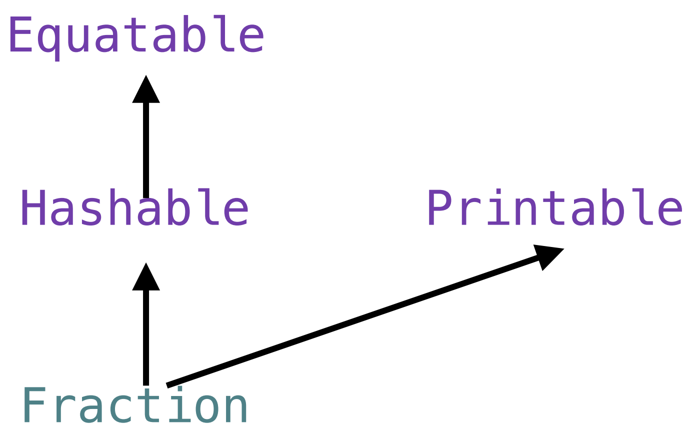
Protocols as abstract classes

Swift

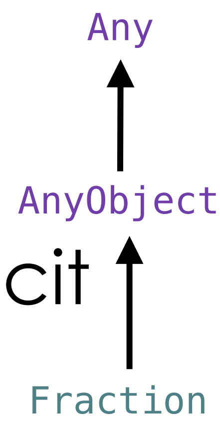
- In Swift, protocols function a bit like abstract classes:

```
class Fraction: Hashable, CustomStringConvertible {
```

- can be read as "Fraction inherits from Hashable and Printable"
- In this interpretation, Swift allows multi-abstract-class inheritance (but still only single non-abstract-class inheritance)



Protocols as abstract classes



Swift

- `AnyObject`—protocol specifying implicit methods that work on all objects
  - Every object conforms to this protocol

```
/// The protocol to which all classes implicitly conform.
/// When used as a concrete type, all known `@objc` methods and
/// properties are available, as implicitly-unwrapped-optional methods
/// and properties respectively, on each instance of `AnyObject`.
@objc protocol AnyObject {
}
```

- `Any`—protocol that doesn't specify any methods
  - Every type conforms to this protocol

```
/// The protocol to which all types implicitly conform
typealias Any = protocol<>
```

Protocols as abstract classes

Swift

- Example: AnyObject—works with object types only

```
func show(x: AnyObject) {
    print(x.description);
}

func isObject(x: AnyObject, sameIntanceAs y: AnyObject) -> Bool {
    if x === y {
        return true
    } else {
        return false
    }
}
```

- Example: Any—works with objects as well as value types

```
func show(x: Any) {
    print(x.description);        ‘Any’ does not have a member named ‘description’
}

func isObject(x: Any, sameIntanceAs y: Any) -> Bool {
    if x === y {        Binary operator ‘===‘ cannot be applied to two ‘Any’ operands
        return true
    } else {
        return false
    }
}
```

Design Pattern - Factory

- Create instances of objects at runtime
  - Usually all have a common interface
- Doesn't expose internal logic
  - Default parameters
- *Builder* doesn't know what object will be created

- Can add complexity

- Plastic toys — different moulds, same factory

Factory - Example

Toolmaker

```
enum Shapes{
    case rectangle, square, triangle, circle
}
enum ShapeFactory{
    static func shape(for s:Shapes) -> Shape{
        switch s {
        case .rectangle:
            return Rectangle()
        case .square:
            return Square()
        case .triangle:
            return Triangle()
        case .circle:
            return Circle()
        }
    }
}
```

```
class Shape{}
class Rectangle : Shape{}
class Square : Rectangle{}
class Triangle: Shape{}
class Circle: Shape{}
```

Builder

```
var myshape = ShapeFactory.shape(for: .circle)
```

Specific Shape selected by user, and not Builder!

Factory in the real world?

Summary?