Polymorphism

In Swift

What is Polymorphism?

To begin to understand this concept, it’s helpful to know that the word polymorphism comes from the greek meaning many shapes (or forms).

poly… = many **…morphism** = shape/form

In the context of Object Oriented Programming, this means:

> “the provision of a single interface to entities of different types.”

Polymorphism allows the expression of common behavior between types of objects which have similar traits.

Inheritance vs Polymorphism

Whilst inheritance is a mechanism for allowing existing code to be reused in a program, polymorphism is considered more to be a mechanism to dynamically decide what form of a function should be invoked.

Inheritance can be thought of as one of the mechanisms we can use to achieve polymorphism in our code via the use of a **hierarchy**. However there are other ways to achieve polymorphism besides inheritance, ways which take a more protocol based approach, and we will explore these other forms of polymorphism later in the presentation (Ref#: H).

Inheritance vs Polymorphism

To expand on the concept further, with Polymorphism, the interface provided by the objects is the same, but what's going on under the hood might vary depending on, perhaps, the message sent, or the type passed to a particular function for example. Typically then the result will be an appropriate type-specific behavior/response.

Subtype polymorphism

This refers to where one class or type, a subclass (or subtype) inherits the properties and behaviors of another it’s superclass (or supertype).

This means that program functions, written to operate on elements of the supertype, can also be used to operate on elements of the subtype.

Static or Compile-Time Polymorphism

This means the ability to have several functions, all **called the same thing**, which the compiler can choose between at Compile-Time. This is depending on the arguments passed, and achieved through through **operator overloading**, or **method / function overloading** (via static or early binding).

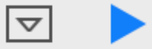
A very simple example of **ad hoc polymorphism** using **operator overloading** is the use of the “+” symbol to denote adding two Integers where Ints are passed in, or alternatively to concatenate two strings when Strings are given.

```
1 + 1 == 2
```

```
"Jane " + "Doe" == “Jane Doe”
```

Static Polymorphism / Method Overloading Example

```
38  struct Van {
39  }
40  struct Tractor {
41  }
42
43  func drive(with vehicle: Van) {
44    print("Driving a Van")
45  }
46
47  func drive(with vehicle: Tractor) {
48    print("Driving a Tractor")
49  }
50
51  drive(with: Tractor())
52  drive(with: Van())
53
```



```
Driving a Tractor
Driving a Van
```

In this code snippet we have created two different functions with the same name “drive()”. Yet we can send different parameter types to drive using the exact same syntax, and the **complier** will **automatically** select for us the correct function to execute based on the type it detects being passed in, and so we don’t need to remember a bunch of different function names like driveVan, driveTractor but can take advantage of this layer of abstraction to make things easier.

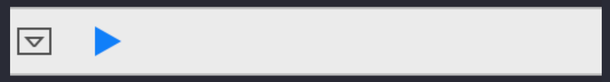
Run-Time or Dynamic Polymorphism

This is polymorphism that happens at run-time and it achieved via method **overriding** (though dynamic or late binding).

Let examine what that actually means now with some examples:

Dynamic Polymorphism / Method Overriding Example

```swift
// 1.
struct Driver {
  let name:String!
  let age: Int!
}
// 2.
class Vehicle {
  var driversName: String?
  // an custom init into which we can inject a Driver
  required init (_ driver: Driver) {
    driversName = driver.name
  }
  func drive() {
    if driversName != nil { print (driversName! + " has started to Drive!") }
  }
}
```



```
Dave has started to Drive a Car!
car cast as Vehicle
```

```swift
// 3.
class Car: Vehicle {
  // in our subclass we may override the implimentation of drive
  override func drive() {
    if driversName != nil { print (driversName! + " has started to Drive a Car!") }
  }
}
// 4.
var me = Driver(name: "Dave", age: 22)
let car = Car(me) // Create a Car by injective a driver
car.drive()
// 5.
if let this = car as Vehicle! {
  print (" 'car' cast as 'Vehicle'")
}
```

1. Create a struct called Driver and give it two required elements of name and age.
2. Create a class called Vehicle and give it a function called drive() and custom init into which we must inject a Driver to create an instance of vehicle.
3. We subclass Vehicle with a class called Car then we choose to **override** the drive function to customize the implementation whilst leaving the interface the same.
4. Now we create an instance of our struct Driver and inject it into an instance of Car called car. When we call car.drive() and see the output: “Dave has started to drive a Car!”
5. We use the Type Casting (coercion) features of Swift to cast (or upcast) our car (which has the type of Car) as type Vehicle, it’s superclass.

Links to Liskov’s Substitution Principle

Thinking about our **SOLID principles**, one crucial principle here, one inadvertently popularized by Barbara Liskov in a talk she once gave, is **Liskov’s Substitution Principle.**

This principle states that if S is a subtype of T, then objects of type T should be substitutable with objects of type S without altering or changing any of the desirable properties that T may have. In other words that **objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program.** If we find ourselves breaking this principle then we have probably failed to correctly identify the right set of abstractions we should be using.

Parametric Polymorphism and Generics

Parametric Polymorphism

This is when code is written without reference to any specific type, and for that reason it can be used transparently with a range of different types.

This is most often known as **generics**, but in the language of functional programming it can be referred to just using the term **polymorphism**.

So a function or a data type can be written generically so that it can handle a range of values irrespective of their type.

Parametric Polymorphism / Generics

Generics may also be used to increase **type safety** by setting expected types during declaration (as these expected types might be specified to be those which adhere to a particular **protocol** like **Comparable**).

Types that conform to the **Comparable** protocol in Swift are types which can be compared using the relational operators <, <=, >=, and >, or where one element of that type might be compared with another using these operators.

Parametric Polymorphism / Generics

To use generics in Swift we use what is known as a ***Generic Parameter Clause*** which is a comma-separated list of generic parameters enclosed in angle brackets (<>) . Each generic parameter taking the form:

```
type parameter: constraint
```

The **type parameter** is simply the name of a **placeholder type** (for example, T, U, V or any word beginning with a capital letter denoting a generic placeholder type).

And the c**onstraint** part is a reference to a type parameter which inherits from a specific class or conforms to a particular protocol (like Comparable) or protocol composition.

Parametric Polymorphism / Generics

So in Swift this could typically look something like this:

```
func mergeSort <T: Comparable>(_ array: [T]) -> [T] {

            guard array.count > 1 else { return array }

            let middleIndex = array.count / 2

            let leftArray = mergeSort(Array(array[0..<middleIndex]))

            let rightArray = mergeSort(Array(array[middleIndex..<array.count]))

            return merge(leftArray, rightArray)

}
```

Conclusion

In conclusion, we have seen what polymorphism means in the context of Object-Oriented programming and different ways this can be used.

The advantages of polymorphism include making code more reusable, and flexible, it can also simplify “the programming interface [permitting] conventions to be established that can be reused in class after class.

Instead of inventing a new name for each new function you add to a program, the same names can be reused. The programming interface can be described as a set of abstract behaviors, quite apart from the classes that implement them.”