Polymorphism

Polymorphism in Kotlin, as in other object-oriented programming languages, refers to the ability of different objects to be accessed through the same interface, allowing a single function or method to behave differently based on the object it is acting upon.

Polymorphism through interfaces, allows objects of different types to be accessed through the same interface reference. This enables code to be written more flexibly and generally, promoting reusability. With polymorphism, the same method can behave differently based on the object it is invoked on, depending on the actual implementation provided by the object.

There are two types of polymorphism in Kotlin:

1. Compile-time Polymorphism (Method Overloading):

- This occurs when multiple methods in the same class have the same name but different parameter lists. The correct method is determined at compile time, based on the number or type of arguments passed.

Example:

```
kotlin
Copy code
class Calculator {
    // Overloaded function to add two integers
    fun add(a: Int, b: Int): Int {
        return a + b
    }

    // Overloaded function to add three integers
    fun add(a: Int, b: Int, c: Int): Int {
        return a + b + c
    }
}

fun main() {
    val calc = Calculator()
    println(calc.add(5, 10))      // Calls add(Int, Int)
    println(calc.add(5, 10, 15))  // Calls add(Int, Int, Int)
}
```

In this example, the `add` function is overloaded, and Kotlin knows which one to call based on the number of arguments.

2. Runtime Polymorphism (Method Overriding):

- This occurs when a subclass overrides a method in its parent class. The decision of which method to invoke is made at runtime based on the actual type of the object, not the reference type.

Example:

```
kotlin
Copy code
open class Animal {
    open fun sound() {
        println("Animal makes a sound")
    }
}

class Dog : Animal() {
    override fun sound() {
        println("Dog barks")
    }
}

class Cat : Animal() {
    override fun sound() {
        println("Cat meows")
    }
}

fun main() {
    val animal: Animal = Dog()
    animal.sound()  // Calls Dog's overridden method, output: Dog barks

    val anotherAnimal: Animal = Cat()
    anotherAnimal.sound()  // Calls Cat's overridden method, output: Cat meows
}
```

Here, the method `sound()` is overridden in the `Dog` and `Cat` classes. At runtime, when the `sound` function is called on the `animal` reference, the actual object's method (either `Dog` or `Cat`) is executed.

Interface Example for Polymorphism

Consider an interface `Shape` with a method `draw()`. Multiple classes (like `Circle`, `Square`, and `Triangle`) implement this interface. The concept of polymorphism allows you to call `draw()` on any `Shape` object, regardless of its concrete class.

Example:

```
kotlin
Copy code
// Defining an interface with a polymorphic method
interface Shape {
    val sides: Int
    fun draw()
}

// Implementing the interface in different classes
class Circle : Shape {
    override val sides: Int = 0
    override fun draw() {
        println("Drawing a Circle with $sides sides")
    }
}

class Square : Shape {
    override val sides: Int
        get()= 4
    override fun draw() {
        println("Drawing a Square $sides sides")
    }
}
```

```
class Triangle : Shape {
    override val sides: Int = 3
    override fun draw() {
        println("Drawing a Triangle $sides sides")
    }
}
```

By using a **common interface type** (`Shape`), you can create an array or list of various shapes and invoke their `draw()` method polymorphically. Each shape will execute its own version of `draw()`, but from the perspective of the code, you treat them as `Shape` objects.

```
fun main() {
    // Creating a list of different shapes
    val shapes: List<Shape> = listOf(Circle(), Square(), Triangle())

    // Looping through the shapes and calling draw on each one
    for (shape in shapes) {
        shape.draw()  // Polymorphism: same method, different behavior
    }
}
```

Output: -

Drawing a Circle with 0 sides.

Drawing a Square with 4 sides.

Drawing a Triangle with 3 sides.