Overriding Members of Open Parent Class

In Kotlin, **overriding** refers to the ability to provide a specific implementation of a method that is already defined in a superclass. When you override a method in a subclass, the subclass version of the method replaces the superclass version when called on an instance of the subclass.

“***Redefining a method of super class in its subclass with same signature***”

Key Points of Overriding in Kotlin:

1. **`open` keyword**: In Kotlin, methods and properties are **final** by default, meaning they cannot be overridden. To allow overriding, the method or property in the superclass must be marked with the `open` keyword.
2. **`override` keyword**: When you provide a new implementation for an open method in the subclass, you must use the `override` keyword to indicate that you are overriding an existing method.
3. **Call to superclass method**: You can use `super` to call the superclass's version of the overridden method.

Syntax Example:

```
// Superclass
open class Animal {
    open fun sound() {
        println("Animal makes a sound")
    }

    fun walk(){// final method can’t be overridden but is inherited
       println(“Animal can walk.”)
    }
}

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

fun main() {
    val myDog = Dog()
    myDog.sound()   // Output: Dog barks
    myDog.walk()    // Output: Animal can walk
}
```

In the above example:

- The `Animal` class has an open method `sound()`, which can be overridden.
- The `Dog` class overrides the `sound()` method and provides its own implementation (`Dog barks`).
- When `sound()` is called on a `Dog` object, the overridden method in `Dog` is executed instead of the one in `Animal`.
- The `Dog` class can’t overrides the `walk()` method, so when `walk()` is called on a `Dog` object, the actual inherited method from `Animal class` executed.

1. Calling the Superclass Method:

If you want to use the original behavior from the superclass in your overridden method, you can call it using `super`.

```
kotlin
Copy code
class Dog : Animal() {
    override fun sound() {
        super.sound()  // Calls the superclass method
        println("Dog barks")
    }
}
```

Overriding allows subclasses to tailor or extend the behavior of their superclass while still maintaining the structure of inheritance in Kotlin.

2. Overriding Properties:

Just like methods, you can also override properties. If a property in the superclass is marked as `open`, the subclass can override it with its own value or implementation.

```
kotlin
Copy code
open class Shape {
    open val sides: Int = 0
}

class Triangle : Shape() {
    override val sides: Int = 3
}

fun main() {
    val triangle = Triangle()
    println(triangle.sides)  // Output: 3
}
```

3. Overriding Getters and Setters:

If the superclass has properties with custom `get` and `set` methods, you can override those in the subclass as well.

```
kotlin
Copy code
open class Rectangle {
    open val area: Int
        get() = 0 // getter in parent returns 0
}

class Square(val side: Int) : Rectangle() {
    override val area: Int
        get() = side * side // getter in child return side * side
}
```

4. Visibility Modifiers:

If the superclass method or property has a more restrictive visibility (`protected`, `internal`, `private`), overriding follows those rules. Overriding methods or properties can increase the visibility of the original member (e.g., overriding a `protected` method with a `public` one is allowed). But decreasing the visibility of a method when overriding it is **not allowed** in Kotlin. When you override a method, the overridden method must have the **same** or **greater** visibility than the method in the superclass

5. Final Methods/Properties:

If a method or property in the superclass is marked with the `final` keyword, it cannot be overridden by subclasses. This is used when you want to prevent any further modification of the method's behavior.

```
kotlin
Copy code
open class BaseClass {
    final fun doNotOverride() {
        println("This cannot be overridden")
    }
}
```

6. Abstract Methods:

When a method is marked as `abstract` in a superclass, the subclass **must** override it to provide its implementation. You'll encounter this with abstract classes.

```
kotlin
Copy code
abstract class Vehicle {
    abstract fun move()
}

class Car : Vehicle() {
    override fun move() {
        println("Car is moving")
    }
}
```

We will learn this more in abstract class session

Key Points about Overriding

- The **argument list** needs to be exactly same as that of overridden method present in base class.

- **Overriding a method by changing only the return type** is generally **not allowed** unless the new return type is a **subtype** of the original return type. This rule follows the principles of **covariant return types**, which means that you can override a method and return a more specific type than the one defined in the superclass.

  Example of Overriding with a Covariant Return Type:

  ```
  kotlin
  Copy code
  open class Animal

  class Dog : Animal()

  open class AnimalShelter {
      open fun getAnimal(): Animal {
          return Animal()
      }
  }

  class DogShelter : AnimalShelter() {
      override fun getAnimal(): Dog {  // Allowed because Dog is a subtype of Animal
          return Dog()
      }
  }

  fun main() {
      val shelter: AnimalShelter = DogShelter()
      val animal = shelter.getAnimal()  // Returns a Dog, but type is Animal
      println(animal is Dog)  // Output: true
  }
  ```

- The access level can’t be set to more restrictive than method in parent class. It can either be **same** or **greater** visibility than the method in the superclass.
- Instance method can only be overridden if they are declared as open.
- Any instance method in parent class is declared final by default so they can’t be overridden, but is inherited.
- Static method can’t be overridden since they are bind at compile time. In Kotlin, **there are no true static methods** like in Java. Instead, Kotlin uses **companion objects** to simulate static behavior. Methods and properties declared inside a companion object can be accessed without creating an instance of the class, similar to how static methods work in Java.

  Creating Static-like Methods in Kotlin:

  To create a static-like method, you declare a method inside a **companion object**. Here's how:

  ```
  kotlin
  Copy code
  class MyClass {
      companion object {
          fun myStaticMethod() {
              println("This is a static-like method in Kotlin")
  ```

```
        }
    }
}

fun main() {
    MyClass.myStaticMethod()  // Output: This is a static-like method in Kotlin
}
```

No, methods inside a companion object **cannot be overridden**. This is because methods in a companion object are essentially treated as **static** functions in Java (although Kotlin doesn't call them "static"). Static methods are tied to the class, not to an instance of the class.

Example of Companion Object in Subclass (Not Overriding):

```
kotlin
Copy code
open class Parent {
    companion object {
        fun staticMethod() {
            println("Static method in Parent")
        }
    }
}

class Child : Parent() {
    companion object {
        fun staticMethod() { // it can be redeclared
            println("Static method in Child")
        }
    }
}

fun main() {
    Parent.staticMethod()  // Output: Static method in Parent
    Child.staticMethod()   // Output: Static method in Child
}
```

- Constructor can't be overridden as it is not possible that parent and child class have same constructed, even more constructor are not inherited
- `Super.method_name()` will call to parent method that is overridden I child explicitly.
- Properties can be overridden same as methods.