Abstract Class

Abstract Class

- **Definition**: An abstract class in Kotlin is a class that cannot be instantiated directly. It serves as a blueprint for other classes. The abstract class can have abstract methods (which do not have a body) and concrete methods (which do have an implementation).
- **Purpose**: Abstract classes are used when you want to provide some common functionality across multiple subclasses but leave the specific implementation of certain behaviors to the subclasses.
- **Key Points**:
  - Declared with the `abstract` keyword.
  - Can have abstract methods (without body) and non-abstract (concrete) methods (with body).
  - Abstract methods must be overridden in the subclasses that extend the abstract class.
  - Cannot be instantiated directly.

```
kotlin
Copy code
abstract class Animal {
    abstract fun makeSound() // Abstract method: no body

    fun sleep() { // Concrete method: has a body
        println("Sleeping...")
    }
}
```

2. Concrete Class

- **Definition**: A concrete class is a regular class that provides complete implementations for all its methods. If a concrete class extends an abstract class, it must provide implementations for all the abstract methods of the parent class.
- **Purpose**: Concrete classes are used when you need a complete, functional class that can be instantiated and used to create objects.
- **Key Points**:
  - Extends the abstract class and implements all the abstract methods.
  - Can also have additional methods or properties specific to its own behavior.
  - Can be initiated directly.

```
kotlin
Copy code
class Dog : Animal() {
    override fun makeSound() { // Implementing abstract method
        println("Bark")
    }
}

fun main() {
    val myDog = Dog() // Concrete class can be instantiated
    myDog.makeSound() // Output: Bark
    myDog.sleep()     // Output: Sleeping...
}
```

1. Abstract Class Can Have Constructors

- Abstract classes can have constructors just like regular classes. These constructors are used to initialize common properties of the abstract class, which can then be shared with subclasses.

```
abstract class Vehicle(val name: String) {
    abstract fun move()  // Abstract method
}

class Car(name: String) : Vehicle(name) {
    override fun move() {
        println("$name is moving on four wheels")
    }
}

fun main() {
    val myCar = Car("Tesla")
    myCar.move()  // Output: Tesla is moving on four wheels
}
```

2. Abstract Properties

- Along with abstract methods, abstract classes can have **abstract properties**. Abstract properties don't have initial values or bodies and must be overridden by the subclass.

```
abstract class Shape {
    abstract val area: Double // Abstract property

    fun printArea() {
        println("The area is $area")
    }
}

class Circle(val radius: Double) : Shape() {
    override val area: Double
        get() = Math.PI * radius * radius // Providing implementation in subclass
}

fun main() {
    val circle = Circle(5.0)
    circle.printArea()  // Output: The area is 78.53981633974483
}
```

3. Abstract Classes vs Interfaces

- Both abstract classes and interfaces allow for method declarations without implementations. However, there are key differences:
    - **Abstract Class**:
        - *Can hold state (i.e., can have properties with initial values).*
        - *Can have constructors.*
        - *A class can only inherit from **one** abstract class (Kotlin supports single inheritance).*
    - **Interface**:
        - *Cannot hold state (though they can have default methods).*
        - *Cannot have constructors.*
        - *A class can implement **multiple interfaces**.*

```
interface Movable {
    fun move()
}

abstract class Animal(val name: String) {
    abstract fun sound()
}

class Dog(name: String) : Animal(name), Movable {
    override fun sound() {
        println("$name barks!")
    }

    override fun move() {
        println("$name runs!")
    }
}
```

4. Concrete Methods in Abstract Class

- An abstract class can have concrete methods that can either be inherited directly by subclasses or overridden if necessary.

```
abstract class Appliance {
    abstract fun operate()

    fun turnOn() { // Concrete Method
        println("The appliance is now ON")
    }
}

class WashingMachine : Appliance() {
    override fun operate() {
        println("Washing clothes")
    }
         //turnOn() is directly inherited
}

fun main() {
    val wm = WashingMachine()
    wm.turnOn()  // Output: The appliance is now ON
    wm.operate() // Output: Washing clothes
}
```

5. Access Modifiers in Abstract Classes

- You can use **visibility modifiers** (`private`, `protected`, `public`, `internal`) with abstract classes, methods, and properties.
    - **Protected members** in an abstract class are only accessible in subclasses.

```
abstract class Device {
    protected abstract fun boot()

    fun startDevice() {

        println("Starting device...")

        boot() // Protected method can be used inside the class
    }
}
```

```
class Computer : Device() {
    override fun boot() {
        println("Booting the computer...")
    }
}

fun main() {
    val pc = Computer()
    pc.startDevice()  // Output: Starting device... Booting the computer...
}
```

6. Multiple Abstract Classes and Interface Inheritance

- A class in Kotlin can only inherit from one abstract class, but it can implement multiple interfaces.

```
//Interface
interface Driveable {
    fun drive()
}

interface Steerable {
       fun steer(direction: String) // Method in the interface
}


//Abstract Class

abstract class Machine(val name: String) {
    abstract fun powerOn()
}


class Car (name: String): Machine(name), Driveable, Steerable {

    override fun powerOn() {
        println("$name Car powered on")
    }

     override fun steer(direction: String){
         println("$name Car is steering towards $direction"))
     }


    override fun drive() {
        println("$name Car is driving on road")
    }
}

fun main() {

    val myCar = Car("Tesla")

    myCar.powerOn()      // Output: Tesla Car powered on

    myCar.drive()        // Output: Tesla Car is driving on road

    myCar.steer("Right") //Output: Tesla Car is steering towards right


}
```

7. Non-Abstract Methods marked as Open Can Be Overridden by Subclasses

- In subclasses of an abstract class, not only are you required to implement abstract methods, but you can also choose to override non-abstract methods if you want to provide a different implementation.

```
abstract class Printer {
    abstract fun printDocument()

    open fun showStatus() {// general overriding
        println("Ready to print")
    }
}
class LaserPrinter : Printer() {
    override fun printDocument() {
        println("Printing document with laser printer")
    }

    override fun showStatus() {
        println("Laser printer is online")
    }
}
```

8. Cannot Have Instances of Abstract Classes

- You cannot create an instance of an abstract class directly. You can only create an instance of its subclass.

```
abstract class Animal {
    abstract fun makeSound()
}

// This is invalid and causes a compile error
// val animal = Animal()

class Dog : Animal() {
    override fun makeSound() {
        println("Bark")
    }
}

// This is valid
val animal2: Animal = Dog()
```

9. Overriding Rules in Abstract Classes

- If a subclass doesn't override all abstract members of an abstract class, then that subclass also becomes **abstract**.

```
abstract class Shape {
    abstract fun draw()
}

abstract class Polygon : Shape() {
    // Doesn't override draw() yet, so Polygon is still abstract
}
```

Key Notes

1. A method that has only been declared without an implementation is called as abstract method. It does not have any body, and have only signature/header.
2. A class can also have abstract properties that can be overridden by subclass.
3. A class which has been declared with abstract keyword is known as abstract class. If a class contains even a single abstract method then the class need to be marked as abstract itself.
4. Abstract class can have concrete (non-abstract) as well as abstract methods & properties.
5. Abstract class can have primary/secondary constructor. But the subclass must first delegate any of its parameter to initialize parent abstract class property.
6. If something is abstract it means it is incomplete and can't be used directly
7. We can not create object of abstract class directly, however we can create reference of abstract class that points to any of its subclass that implements all of its abstract methods.
8. Generally, speaking a subclass should provide implementation of abstract class but if it does not the subclass also need to be marked as abstract.
9. Since, abstract class or abstract method needs to be implemented by subclass so by default they are open we need not to mark them open.
10. Any other concrete method/property present in abstract class need to mark open if we want to override them in child class along with those abstract method/properties.
11. During overriding some abstract/non-abstract method/property we will use override keyword.
12. If we want to restrict any f/n to get override accidently we can define them in abstract class without open keyword so they are. public finally fun fun_Name([parameter]){ //body }
13. Abstract class can have static method that not linked to any instance, but it is done by the concept of companion object.