# Class

## Class

## What is a Class?

A **class** in Kotlin is a blueprint for creating objects. It can have:

- **Properties** (fields/variables)
- **Functions** (methods)
- **Constructors**
- **Initialization blocks**
- **Nested/inner classes**
- **Inheritance and interfaces**

---

## Basic Class Example

```kotlin
class Person {
    var name: String = "Unknown"
    var age: Int = 0

    fun introduce() {
        println("Hi, I'm $name and I'm $age years old.")
    }
}

fun main() {
    val p = Person()
    p.name = "Alice"
    p.age = 25
    p.introduce()
}
```

---

## Primary Constructor

Kotlin allows concise constructor declarations:

```
class Person(val name: String, var age: Int) {
    fun greet() {
        println("Hello, my name is $name.")
    }
}

fun main() {
    val person = Person("Bob", 30)
    person.greet()
}
```

> `val` makes a read-only property, `var` makes it mutable.

---

## Initializer Block (`init`)

Runs when an object is created:

```
class Car(val brand: String, val year: Int) {
    init {
        println("Car: $brand, Year: $year created.")
    }
}

fun main() {
    val car = Car("Toyota", 2023)
}
```

---

## Secondary Constructor

You can define additional constructors:

```
class Student {
    var name: String
    var age: Int

    constructor(name: String, age: Int) {
        this.name = name
        this.age = age
    }
}
```

---

## Inheritance

Use `open` to allow a class to be inherited.

```
open class Animal {
    fun eat() = println("Eating...")
}

class Dog : Animal() {
    fun bark() = println("Barking...")
}
```

---

## Data Class

For classes used to hold data. Kotlin auto-generates `toString()`, `equals()`, `hashCode()`, `copy()`.

```
data class User(val name: String, val age: Int)

fun main() {
    val u1 = User("Alice", 25)
    println(u1)  // User(name=Alice, age=25)
}
```

---

## Object Declaration (Singleton)

Kotlin makes it easy to create singletons:

```
object Database {
    fun connect() = println("Connected to DB")
}

fun main() {
    Database.connect()
}
```

---

## Nested and Inner Classes

```kotlin
class Outer {
    private val message = "Hello"

    class Nested {
        fun nestedHello() = "Nested Hello"
    }

    inner class Inner {
        fun innerHello() = "Inner says: $message"
    }
}
```

## OOP

Absolutely! Let's break down **all major OOP concepts** with **Kotlin** examples, step by step:

---

## 1. Class and Object

**Class** is a blueprint; **Object** is an instance of a class.

### Kotlin Example:

```kotlin
class Person(val name: String, var age: Int) {
    fun introduce() {
        println("Hi, I'm $name and I'm $age years old.")
    }
}

fun main() {
    val person = Person("Alice", 25)
    person.introduce()
}
```

---

## 2. Inheritance

Allows a class to inherit features (properties and methods) from another class.

### Kotlin Example:

```kotlin
open class Animal {
    fun eat() {
        println("Animal is eating")
    }
}

class Dog : Animal() {
    fun bark() {
        println("Dog is barking")
    }
}

fun main() {
    val dog = Dog()
    dog.eat()  // Inherited from Animal
    dog.bark() // Dog's own method
}
```

> 🔑 Note: Use `open` to make a class inheritable in Kotlin.

---

## 3. Encapsulation

Hides internal state and requires all interaction through an object's methods (protects internal data).

### Kotlin Example:

```kotlin
class Account {
    private var balance: Double = 0.0

    fun deposit(amount: Double) {
        if (amount > 0) balance += amount
    }

    fun withdraw(amount: Double) {
        if (amount > 0 && amount <= balance) balance -= amount
    }

    fun getBalance(): Double {
        return balance
    }
}

fun main() {
    val account = Account()
    account.deposit(500.0)
    account.withdraw(100.0)
    println("Balance: ${account.getBalance()}")
}
```

---

## 4. Abstraction

Hides complex implementation and shows only essential details.

### Kotlin Example with `abstract class`:

```kotlin
abstract class Vehicle {
    abstract fun start()
}

class Car : Vehicle() {
    override fun start() {
        println("Car is starting")
    }
}

fun main() {
    val car: Vehicle = Car()
    car.start()
}
```

> `abstract` classes can't be instantiated directly.

---

## 5. Polymorphism

Same function name behaves differently in different classes.

### a) **Method Overriding** (Run-time Polymorphism):

```kotlin
open class Shape {
    open fun draw() {
        println("Drawing a shape")
    }
}

class Circle : Shape() {
    override fun draw() {
        println("Drawing a circle")
    }
}

fun main() {
    val shape: Shape = Circle()
    shape.draw()  // Output: Drawing a circle
}
```

### b) **Method Overloading** (Compile-time Polymorphism):

```kotlin
class Calculator {
    fun add(a: Int, b: Int): Int {
        return a + b
    }

    fun add(a: Double, b: Double): Double {
        return a + b
    }
}

fun main() {
    val calc = Calculator()
    println(calc.add(5, 3))         // Output: 8
    println(calc.add(2.5, 4.3))     // Output: 6.8
}
```

---

## 6. Interface

Defines a contract that implementing classes must follow.

## Kotlin Example:

```kotlin
interface Drivable {
    fun drive()
}

class Truck : Drivable {
    override fun drive() {
        println("Truck is driving")
    }
}

fun main() {
    val vehicle: Drivable = Truck()
    vehicle.drive()
}
```

> Kotlin allows classes to implement multiple interfaces.