**Object-Oriented Programming (OOP) with Kotlin**

**Encapsulation**

* **Definition:** Encapsulation is the process of wrapping data (variables) and code (methods) together as a single unit. In OOP, it restricts direct access to some of an object’s components, promoting data hiding and security.
* **Implementation in Kotlin:**
  + Kotlin uses **access modifiers** (private, protected, internal, and public) to control visibility.
  + Properties in Kotlin can have different access levels for getters and setters.

**Example:**

class BankAccount(private var balance: Double) {

fun deposit(amount: Double) {

if (amount > 0) balance += amount

}

fun getBalance(): Double {

return balance

}

}

fun main() {

val account = BankAccount(100.0)

account.deposit(50.0)

println("Current Balance: ${account.getBalance()}")

}

* **Benefits:**
  + **Data Protection:** Prevents unauthorized access.
  + **Flexibility:** Allows changing internal implementation without affecting external code.
  + **Improved Maintainability:** Promotes cleaner and more maintainable code.

**Inheritance**

* **Definition:** Inheritance allows a class (child/subclass) to inherit properties and behaviors from another class (parent/superclass), promoting code reuse and hierarchy.
* **Implementation in Kotlin:**
  + Classes are **final** by default in Kotlin. To enable inheritance, mark the superclass with the open keyword.
  + Use : to inherit from a class.

**Example:**

open class Animal(val name: String) {

open fun makeSound() {

println("$name makes a sound")

}

}

class Dog(name: String): Animal(name) {

override fun makeSound() {

println("$name barks")

}

}

fun main() {

val dog = Dog("Buddy")

dog.makeSound()

}

* **Benefits:**
  + **Code Reusability:** Reuse existing class functionalities.
  + **Extensibility:** Easily extend existing behaviors.
  + **Improved Organization:** Helps in logically structuring code.

**Polymorphism**

* **Definition:** Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single interface to represent different types.
* **Implementation in Kotlin:**
  + Achieved through **method overriding** and **interfaces**.
  + Supports **runtime** (dynamic) and **compile-time** (static) polymorphism.

**Example:**

open class Shape {

open fun draw() {

println("Drawing a shape")

}

}

class Circle: Shape() {

override fun draw() {

println("Drawing a circle")

}

}

class Square: Shape() {

override fun draw() {

println("Drawing a square")

}

}

fun main() {

val shapes: List<Shape> = listOf(Circle(), Square())

for (shape in shapes) {

shape.draw()

}

}

* **Benefits:**
  + **Flexibility:** Write generic code that works with different objects.
  + **Scalability:** Add new behaviors without modifying existing code.
  + **Ease of Maintenance:** Simplifies code management.

**Abstraction**

* **Definition:** Abstraction focuses on exposing only essential features while hiding implementation details. It helps in managing complexity by reducing the information to only what’s necessary.
* **Implementation in Kotlin:**
  + Achieved using **abstract classes** and **interfaces**.
  + **Abstract classes** can have both abstract and concrete methods.
  + **Interfaces** define contracts without implementation (can have default methods).

**Example:**

abstract class Vehicle {

abstract fun drive()

}

class Car: Vehicle() {

override fun drive() {

println("Driving a car")

}

}

interface Flyable {

fun fly()

}

class Airplane: Vehicle(), Flyable {

override fun drive() {

println("Taxiing on runway")

}

override fun fly() {

println("Flying in the sky")

}

}

fun main() {

val car = Car()

car.drive()

val airplane = Airplane()

airplane.drive()

airplane.fly()

}

* **Benefits:**
  + **Simplifies Complexity:** Focus on relevant details.
  + **Increased Flexibility:** Define common interfaces without dictating implementation.
  + **Enhanced Security:** Hides unnecessary internal workings.

**Conclusion**

Understanding and applying the four pillars of Object-Oriented Programming—Encapsulation, Inheritance, Polymorphism, and Abstraction—in Kotlin enhances code reusability, maintainability, and scalability. Kotlin’s features, such as concise syntax and powerful type system, make implementing OOP concepts both efficient and intuitive.