Four pillars object-oriented programming (OOP)

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

**Classes:**

1. A class is a blueprint for the objects which have common properties & behaviour. Kotlin classes are declared using **class** keyword.
2. Kotlin class has a class **header** which specifying its type parameters, the primary constructor etc and the class body surrounded by curly braces. Both the header and the body are optional; if the class has no body, the curly braces can be omitted.
3. Class can have can have a **primary** constructor(which will be in header) and any number of **secondary** constructor but make sure when you creating object through secondary constructor than primary constructor should be call from secondary constrictor.
4. If the primary constructor does not have any annotations or visibility modifiers, the **constructor** keyword can be omitted:
5. Kotlin does not(eliminate) have a **new** keyword for object and To create an instance of a class, call the constructor as if it were a regular function:
6. The primary constructor cannot contain any code. Initialization code can be placed in initializer blocks prefixed with the init keyword. and there will be any number Init block and During the initialization of an instance, the initializer blocks are executed in the same order as they appear in the class body
7. If the constructor has annotations or visibility modifiers, the constructor keyword is required and the modifiers go before it:
8. A class can also declare secondary constructors, which are prefixed with constructor:
9. by default all the Kotlin classes are public
10. On the JVM, if all of the primary constructor parameters have default values, the compiler will generate an additional parameterless constructor which will use the default values. This makes it easier to use Kotlin with libraries such as Jackson or JPA that create class instances through parameterless constructors

**class** Person //without header & body

**class** Person { /\*...\*/ } // With body

**class** Person **constructor**(firstName: String) { /\*...\*/ } // with deader & body

**class** Person **constructor**(withDefaultVal: String = “Na”) { /\*...\*/

//secondary contructor (it is mandatory to call parrent contructor from child)

constructor(name: String, ageX: Int) : Person(firstName: name) {}

}

**class** Customer **public** **constructor**(name: String) { /\*...\*/ }

val firstProperty = "First property”;

var firstProperty = "First property”;

init {

println("First initializer block that prints ${name}")

}

init {

println("First initializer block that prints ${name}")

}

**val** person = Person()

val person = Person("Joe Smith”)

val person = Person("Joe Smith”, 39)

# Object:

1. The objects are created from the Kotlin class and they share the common properties and behaviours defined by a class in form of data members (properties) and member functions (behaviours) respectively.
2. every entity has it’s own state, behaviour and identity is known as an object
3. An object is an instance of a class
4. Object Definitions:

* An object is a real-world entity.
* An object is a runtime entity.
* The object is an entity which has state and behavior.
  + State: represents the data (value) of an object
  + Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
* The object is an instance of a class.

1. Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.
2. Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

**Abstraction class:**

1. **Abstraction** is the concept of object-oriented programming that “shows” only essential attributes and “hides” unnecessary information.
2. The main purpose of abstraction is hiding the unnecessary details from the users and show only essential information.
3. In Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user.
4. Abstraction in Kotlin could be achieved by following ways :
   1. Abstract class
   2. Interface

**Example:** like car engine startup process is hidden feature but staring and key on/off is essential feature which one is showing to user.

**Abstract class:**

1. In Kotlin, abstract class is declared using the **abstract** keyword in front of class. An abstract class can not instantiated means we can not create object for the abstract class.
2. **A**bstract class are partial implemented class & it’s declared using the  abstract keyword in front of class.
3. An abstract member does not have an implementation in its class. You don't need to annotate abstract classes or functions with open.
4. An abstract class contains abstract or non-abstract member variables and functions.
5. Abstract classes are partially defined classes, methods and properties which are no implementation but must be implemented into derived class. If the derived class does not implement the properties of base class then is also meant to be an abstract class.
6. Abstract class or abstract function does not need to annotate with open keyword as they are open by default.
7. Abstract member function does not contain its body. The member function cannot be declared as abstract if it contains body in abstract class.
8. We can’t create an object for abstract class.
9. All the variables (properties) and member functions of an abstract class are by default non-abstract. So, if we want to override these members in the child class then we need to use **open** keyword.
10. If we declare a member function as abstract then we does not need to annotate with **open** keyword because these are open by default.
11. An abstract member function doesn’t have a body, and it must be implemented in the derived class.
12. You can override a non-abstract member in child class by using open keyword on non abstract member
13. Abstract Class cannot be initiated but could be extended as a Super Class.
14. The members (properties and methods) of an abstract class are non-abstract unless you explictly use abstract keyword to make them abstract.
15. We can’t create an object for abstract class.
16. All the variables (properties) and member functions of an abstract class are by default *non-abstract*. So, if we want to override these members in the child class then we need to use **open** keyword.
17. If we declare a member function as abstract then we does not need to annotate with **open** keyword because these are open by default.
18. An abstract member function doesn’t have a body, and it must be implemented in the derived class.

**abstract class Employee(val name: String,val experience: Int) {**  // Non-Abstract

// Abstract Property (Must be overridden by Subclasses) and will not be initialise value

abstract var salary: Double

// Abstract Methods (Must be implemented by Subclasses) and it’s not have a body

abstract fun dateOfBirth(date:String)

// Non-Abstract Method

open fun employeeDetails() {

println("Name of the employee: $name")

}

}

**class Engineer(name: String,experience: Int) : Employee(name,experience) {**

override var salary = 500000.00

override fun dateOfBirth(date:String){

println("Date of Birth is: $date")

}

open fun employeeDetails() {

println("Name of the employee: $name")

}

}

fun main(args: Array<String>) {

val eng = Engineer("Praveen",2)

eng.employeeDetails()

eng.dateOfBirth("02 December 1994")

}

**Interface:**

1. Interfaces in Kotlin can contain declarations of abstract methods, as well as method implementations. What makes them different from abstract classes is that interfaces cannot store a state. They can have properties, but these need to be abstract or provide accessor implementations.
2. An interface is defined using the keyword interface:
3. Interfaces are custom types provided by Kotlin that cannot be instantiated directly.
4. Using interface supports functionality of multiple inheritance.
5. It can be used achieve to loose coupling.
6. The methods which are only declared without their method body are abstract by default.
7. You can declare properties in interfaces. A property declared in an interface can either be abstract or provide implementations for accessors.
8. An interface can derive from other interfaces, meaning it can both provide implementations for their members and declare new functions and properties. Quite naturally, classes implementing such an interface are only required to define the missing implementations:

**interface MyInterface {**

fun bar()

fun foo() {

// optional body

}

}

**class Child : MyInterface {**

override fun bar() {

// body

}

override fun foo() {

// body

}

init{

//call super method from init block

super.foo();

super.bar();

}

func callParentMethod(){

Child child = Child()

Child.foo();

Child.bar();

}

}

————-

**interface MyInterface2: MyInterface{**

fun Eat()

fun Eat2(){

// optional body

}

override fun bar(){

}

override fun foo() {

// optional body

}

}

**class Child2 : MyInterface2 {**

**//method / function** bar() & foo() are optional

override fun bar() {

// body

}

override fun foo() {

// body

}

**//method / function** Eat() & Eat2() are required

override fun Eat() {

// body

}

override fun Eat2() {

// body

}

func callParentMethod(){

Child2 child = Child2()

Child.foo();

Child.bar();

}

}

|  |  |
| --- | --- |
| Interfaces | Abstract Classes |
| Cannot have initialized variables. | Can have initialized variables. |
| A single class can implement multiple interfaces. | For a class, only one abstract class could be a super class. |
| **interface** keyword is used to define an interface. | **abstract** keyword is used to define an abstract class. |

|  |  |
| --- | --- |
| The interface can have only abstract methods. | An abstract class can have both abstract and non-abstract methods. |
| It supports multiple inheritances. | It does not support multiple inheritances. |
| It can not provide the implementation of the abstract class. | It can provide the implementation of the interface. |
| An interface can have only have public abstract methods. | An abstract class can have protected and abstract public methods. |
| The interface can only have a public static final variable. | An abstract class can have final, static, or static final variable with any access specifier. |

**Encapsulation:** Encapsulation is **a mechanism of wrapping the data (variables) and code acting on the data (methods/function) together/into a single unit**. In encapsulation, the variables of a class will be hidden from other classes by making it private, and can be accessed only through the methods of their current class by making public.

* Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
* As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
* It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.
* Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

**Polymorphism: Poly(Many) + Morphism(Form)**

**Polymorphism is greek word whose meaning is “Same Object having different behaviour”** Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways.

**Real life example of polymorphism:** A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.”

1. **Compile time Polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by **function overloading. in compile time polymorphism Method resolution are done by compiler.** 
   1. **Method Overloading**: When there are multiple method/functions with same name but different type / no of parameters in same class than these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

2. **Runtime Polymorphism:** It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. **in run time polymorphism Method resolution done by JVM.**

1. **[Method overriding](https://www.geeksforgeeks.org/overriding-in-java/):** on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden. it is achieve between two class(parent & child) , means -> when we declare parent class method in child class with same name & parameter is name as overriding.

**Rules for Java method overriding**

* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the subclass cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* Constructors cannot be overridden

|  |  |  |
| --- | --- | --- |
| Sr.No | **Compile Time Polymorphism** | **Run time Polymorphism** |
| 1 | In Compile time Polymorphism, the call is resolved by the compiler. | In Run time Polymorphism, the call is not resolved by the compiler. |
| 2 | It is also known as Static binding, Early binding and overloading as well. | It is also known as Dynamic binding, Late binding and overriding as well. |
| 3 | Method overloading is the compile-time polymorphism where more than one methods share the same name with different parameters or signature and different return type. | Method overriding is the runtime polymorphism having same method with same parameters or signature, but associated in different classes. |
| 4 | It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and pointers. |
| 5 | It provides fast execution because the method that needs to be executed is known early at the compile time. | It provides slow execution as compare to early binding because the method that needs to be executed is known at the runtime. |
| 6 | Compile time polymorphism is less flexible as all things execute at compile time. | Run time polymorphism is more flexible as all things execute at run time. |

fun main() {

println("Hello, world!!!")

lateinit var myFullName: String;

val srName: String = "Rai"

val firstName: String = "Abhishek";

var age: String? = "18"

//age = null

val getFullName: String by lazy{

"Abhishek Rai"

}

println("MyFull Name: ${getFullName} ")

println("ORG Name: ${MyCompany.myORG.name} ")

myFullName = getFullName

println("My lateInit value is: ${myFullName}")

var lenghtX = age?.length

var lxxx = age!!.length

var lenx = if(age != null) age.length else -1

println(lenx)

val parm: Any = true

if(parm is Int){

println("Given No is Interger")

}else if(parm is String){

println("Given No is String")

}else if(parm is Boolean){

println("Given No is Boolean")

}else{

println("Given No is Any")

}

var myObj1 = MySingletonClass;

var myObj2 = MySingletonClass;

println("Singleton Object: ${myObj1}")

println("Singleton Object: ${myObj2}")

var myObj11 = MyCompany();

var myObj22 = MyCompany();

println("Class Object: ${myObj11}")

println("Class Object: ${myObj22}")

val dta1 = MyData("Abhi")

val dta2 = MyData("Abhi", "CKUser")

println("My Data Name: ${dta1.userName} and Pass: ${dta1.userPass}")

for (enumData in MyEnum.values()) {

println(enumData.name)

}

var selaXX = MySealeadClass.A("OK");

println("${selaXX.name}")

var abs = AbstractImpletion();

println("${abs.firstName}")

val vModel: MyViewModel by viewModel();

vModel.loadUserData.obserbe(this, dta->{

})

}

class MyCompany {

companion object myORG{

const val name = "Abhioshek"

}

}

data class MyData(val userName: String, var userPass: String = "Abh@321"){

init{

userPass = userPass + "OK"

}

}

object MySingletonClass{

var myName: String = "CK";

init{

myName = "CK Rai"

}

}

enum class MyEnum {

Success,

Failed,

NotFound

}

sealed class MySealeadClass{

data class A(var name: String) : MySealeadClass()

class B(var Age: Int): MySealeadClass()

}

abstract class MyAbstrack{

abstract var firstName: String

abstract fun getFullName()

}

interface MyInterface{

fun MyComp(){

println("MyInterface: @MyComp")

}

abstract fun MyCompName()

}

class AbstractImpletion: MyAbstrack(), MyInterface{

override var firstName = "Jiya"

override fun getFullName(){

println("Method overriden")

}

override fun MyComp(){

println("Method MyComp")

}

override fun MyCompName(){

println("Method MyCompName")

}

}

class MyViewModel: ViewModel(){

private val getUserData: MutableLiveData<List<MyData>> by lazy{

MyData("List1", "List2")

}

func loadUserData():MutableLiveDtaa<List<MyData>>{

return getUserData;

}

}

**Serialization V/s Deserialization**

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory.

**Points to remember**

1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.

2. Only non-static data members are saved via Serialization process.

3. Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.

4. Constructor of object is never called when an object is deserialized.

5. Associated objects must be implementing Serializable interface.

**class** Demo **implements** java.io.Serializable {

**public** **int** a;

**public** String b;

    // Default constructor

**public** Demo(**int** a, String b)

    {

**this**.a = a;

**this**.b = b;

    }

}

Demo object = **new** Demo(1, "geeksforgeeks");

String filename = "file.ser";

 // Serialization

**try**{

     //Saving of object in a file

     FileOutputStream file = **new** FileOutputStream(filename);

     ObjectOutputStream out = **new** ObjectOutputStream(file);

      // Method for serialization of object

       out.writeObject(object);

       out.close();

       file.close();

       System.out.println("Object has been serialized");

     }

**catch**(IOException ex){

         System.out.println("IOException is caught");

     }

// Deserialization

Demo object1 = **null**;

**try**

   {

      // Reading the object from a file

String filename = "file.ser";

  FileInputStream file = **new** FileInputStream(filename);

  ObjectInputStream in = **new** ObjectInputStream(file);

// Method for deserialization of object

object1 = (Demo)in.readObject();

in.close();

file.close();

System.out.println("Object has been deserialized ");

System.out.println("a = " + object1.a);

System.out.println("b = " + object1.b);

}

**catch**(IOException ex){

  System.out.println("IOException is caught");

}

**Comparable vs Comparator**

**Comparable:** As the name itself suggests, **Comparable** is an [interface](https://www.edureka.co/blog/java-interface/) which defines a way to compare an object with other objects of the same type. It helps to sort the objects that have self-tendency to sort themselves, i.e., the objects must know how to order themselves. **Eg:** Roll number, age, salary. This interface is found in *java.lang package* and it contains only one method, i.e., *compareTo().*

Java Comparator is **an interface for sorting Java objects**. Invoked by “java. util. comparator,” Java Comparator compares two Java objects in a “compare(Object 01, Object 02)” format. Using configurable methods, Java Comparator can compare objects to return an integer based on a positive, equal or negative comparison.

**Comparable v/s Comparator in Java**

|  |  |
| --- | --- |
| Comparable in Java | Comparator in Java |
| Comparable interface is used to sort the objects with natural ordering. | Comparator in Java is used to sort attributes of different objects. |
| Comparable interface compares “this” reference with the object specified. | Comparator in Java compares two different class objects provided. |
| Comparable is present in java.lang package. | A Comparator is present in the java.util package. |
| Comparable affects the original class, i.e., the actual class is modified. | Comparator doesn’t affect the original class |
| Comparable provides compareTo() method to sort elements. | Comparator provides compare() method, equals() method to sort elements. |