Class & Object

Basic Syntax:

```
class myClass { // class Header
   // class Body
}
```

Example:

```
class myClass {
      // property (data member)
   private var name: String = "Anthony"
     // member function
   fun printMe() {
      print("My Named is : "+name)
   }
}
fun main(args: Array<String>) {
   val obj = myClass() // create obj object of myClass class
   obj.printMe()
}
```

Nested Class

By definition, when a class has been created inside another class, then it is called as a nested class. In Kotlin, nested class is by default static, hence, it can be accessed without creating any object of that class. In the following example, we will see how Kotlin interprets our nested class.

```
fun main(args: Array<String>) {
      val demo = Outer.Nested().foo() // calling nested class method
      print(demo)
}
class Outer {
   class Nested {
      fun foo() = "Welcome to Nested Class"
   }
}
```

Inner Class

When a nested class is marked as a “inner”, then it will be called as an Inner class.

An inner class can be accessed by the data member of the outer class. In the following example, we will be accessing the data member of the outer class.

```
fun main(args: Array<String>) {
   val demo = Outer().Nested().foo() // calling nested class method
   println(demo)
}
class Outer {
   private val welcomeMessage: String = "Welcome to the InnerClass Message"
   inner class Nested {
      fun foo() = welcomeMessage
   }
}
```

Anonymous Inner Class

Anonymous inner class is a pretty good concept that makes the life of a programmer very easy. Whenever we are implementing an interface, the concept of anonymous inner block comes into picture. *The concept of creating an object of interface using runtime object reference is known as anonymous class*. In the following example, we will create an interface and we will create an object of that interface using Anonymous Inner class mechanism.

```
interface Human {
   fun think()
}

fun main(args: Array<String>) {
     var programmer : Human = object:Human {
                    // creating an instance of the interface
          override fun think() {
                    // overriding the think method
           print("I am an example of Anonymous Inner Class ")
          }
     }
     programmer.think()
}
```

Constructors

➔ Kotlin has **two types** of constructor -

- one is the primary constructor and
- the other is the secondary constructor.

➔ One Kotlin class can have one primary constructor, and one or more secondary constructor.

➔ Java constructor initializes the member variables, however, in Kotlin the primary constructor initializes the class, whereas the secondary constructor helps to include some extra logic while initializing the same.

➔ The primary constructor can be declared at class header level as shown in the following example.

```
class Person(val firstName: String, var age: Int) {

  // class body

}
```

```
fun main(args: Array<String>) {
     val person1 = Person("Jainul", 35)
     println("First Name = ${person1.firstName}")
     println("Age = ${person1.age}")
}
class Person(val firstName: String, var age: Int) {
}
```

The primary constructor cannot contain any code.
Initialization code can be placed in initializer blocks, which are prefixed with the init keyword.

```
class InitOrder(name: String) {
     val firstProperty = "First property:
                                        $name".also(::println)
    init {
        println("First initializer block that prints ${name}")
    }

    val secondProperty = "Second property:
                          ${name.length}".also(::println)
    init {
        println("Second initializer block that prints $
{name.length}")
    }
}

fun main() {
    InitOrder("Jainul")
}
```

Output:

```
First property: Jainul
First initializer block that prints Jainul
Second property: 6
Second initializer block that prints 6
```

Note that parameters of the primary constructor can be used in the initializer blocks.

They can also be used in property initializers declared in the class body:

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

In fact, for declaring properties and initializing them from the primary constructor, Kotlin has a concise syntax:

```
class Person(val firstName: String, val lastName: String, var age: Int) { ... }
```

Much the same way as regular properties, the properties declared in the primary constructor can be mutable (**var**) or read-only (**val**).

If the constructor has annotations or visibility modifiers, the constructor keyword is required, and the modifiers go before it:

```
class Customer public @Inject constructor(name: String) { ... }
```

Secondary Constructors

The class can also declare secondary constructors, which are prefixed with constructor:

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

If the class has a primary constructor, *each secondary constructor needs to delegate to the primary constructor*, either directly or indirectly through another secondary constructor(s).

Delegation to another constructor of the same class is done using the **this** keyword:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Note that code in initializer blocks effectively becomes part of the primary constructor.
Delegation to the primary constructor happens as the first statement of a secondary constructor, so the code in all initializer blocks is executed before the secondary constructor body. Even if the class has no primary constructor, the delegation still happens implicitly, and the initializer blocks are still executed:

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor")
    }
}

fun main() {
    Constructors(1)
}
```

https://kotlinlang.org/docs/reference/classes.html

- If a **non-abstract class** does not declare any constructors (primary or secondary), it will have a generated primary constructor with no arguments.
- The visibility of the constructor will be **public**.
- If you ***do not want*** your class to have a public constructor, you need to declare an empty primary constructor with non-default visibility:

```
class DontCreateMe private constructor () { ... }
```

Creating instances of classes

To create an instance of a class, we call the constructor as if it were a regular function:

Example:

```
val invoice = Invoice()
val customer = Customer("Joe Smith")
```

Note: that Kotlin does not have a *new* keyword.

Creating instances of nested, inner and anonymous inner classes is described in Nested classes.

Class Members

Classes can contain:

- Constructors and initializer blocks
- Functions
- Properties
- Nested and Inner Classes
- Object Declarations

Inheritance

All classes in Kotlin have a common ***superclass*** ***Any***, that is the default ***superclass*** for a class with no supertypes declared:

```
class Example // Implicitly inherits from Any
```

Note: ***Any*** is not java.lang.Object; in particular, it does not have any members other than equals(), hashCode() and toString(). Please consult the Java interoperability section for more details.

Inheritance is an important feature of object oriented programming language.
Inheritance allows to inherit the feature of existing class (or ***base*** or ***parent*** class) to new class (or *derived* class or *child* class).
The main class is called super class (or parent class) and the class which inherits the superclass is called subclass (or child class).

- The subclass contains features of superclass as well as its own.

The concept of inheritance is allowed when two or more classes have same properties. It allows code reusability.

A derived class has only one base class but may have multiple interfaces whereas a base class may have one or more derived classes.

In Kotlin, the derived class inherits a base class using: operator in the class header (after the derive class name or constructor)

```
open class Base(p: Int){

}
class Derived(p: Int) : Base(p){

}
```

Suppose that,we have two different classes "***Programmer***" and "***Salesman***" having the common properties 'name','age', and 'salary' as well as their own separate functionalities **doProgram()** and **fieldWork()**.
The feature of inheritance allows that we can inherit (Employee) containing the common features.

```
open class Employee(name: String, age: Int, salary: Float) {
    // code of employee
}

class Programmer(name: String, age: Int, salary: Float):
                                    Employee(name,age,salary) {
    // code of programmer
}

class Salesman(name: String, age: Int, salary: Float):
                                Employee(name,age,salary) {
    // code of salesman
}
```

All Kotlin classes have a common superclass "Any".
It is a default superclass for a class with no supertypes explicitly specified.

Kotlin *open* keyword

As Kotlin classes are final by default, they cannot be inherited simply.

We use the *open* keyword before the class to inherit a class and make it to non-final,

For example:

```
open class Example{
// I can now be extended!
}
```

For example:

```
open class Base{
     val x = 10
}
class Derived: Base() {
    fun foo() {
          println("x is equal to " + x)
    }
}
fun main(args: Array<String>) {
     val derived = Derived()
     derived.foo()
}
```

Kotlin Inheriting methods from a class

```
open class Bird {
     fun fly() {
          println("flying...")
     }
}
class Duck: Bird() {
     fun swim() {
          println("swimming...")
     }
}
fun main(args: Array<String>) {
     val duck = Duck()
     duck.fly()
     duck.swim()
}
```

Kotlin Inheritance Example

Here, we declare a class Employee is superclass and *Programmer* and *Salesman* are their subclasses. The subclasses inherit properties name, age and salary as well as subclasses containtheir own functionalities like doProgram() and fieldWork().

Example:

```
open class Employee(name: String, age: Int, salary: Float) {
     init{
          println("Name is $name.")
          println("Age is $age")
          println("Salary is $salary")
     }
}
class Programmer(name: String, age: Int, salary: Float):
                                        Employee(name,age,salary){
    fun doProgram() {
println("programming is my passion.")
    }
}
class Salesman(name: String, age: Int, salary: Float):
                                        Employee(name,age,salary){
    fun fieldWork() {
          println("travelling is my hobby.")
    }
}
fun main(args: Array<String>){
     val obj1 = Programmer("Ashu", 25, 40000f)
     obj1.doProgram()
     val obj2 = Salesman("Ajay", 24, 30000f)
     obj2.fieldWork()
}
```

Output:

```
Name is Ashu.
Age is 25
Salary is 40000.0
programming is my passion.
Name is Ajay.
Age is 24
Salary is 30000.0
travelling is my hobby.
```

Kotlin Inheritance and primary constructor

If the base and derived class both having primary constructor in that case the parameters are initialized in the primary constructor of base class.

In above example of inheritance, all classes contain three parameters "*name*", "*age*" and "*salary*" and all these parameters are initialized in primary constructor of base class.

When a base and derived class both contains different numbers of parameters in their primary constructor then base class parameters are initialized form derived class object.

For example:

```
open class Employee(name: String,salary: Float) {
     init {
          println("Name is $name.")
          println("Salary is $salary")
     }
}
class Programmer(name: String, dept: String, salary: Float):
                                        Employee(name,salary){
     init{
          println("Name $name of department $dept
                         with salary $salary.")
     }
     fun doProgram() {
          println("Programming is my passion.")
     }
}
class Salesman(name: String, dept: String, salary: Float):
                                        Employee(name,salary){
     init{
          println("Name $name of department $dept
                              with salary $salary.")
     }
    fun fieldWork() {
          println("Travelling is my hobby.")
    }
}
```

```
fun main(args: Array<String>){
     val obj1 = Programmer("Ashu", "Development", 40000f)
     obj1.doProgram()
     println()
     val obj2 = Salesman("Ajay", "Marketing", 30000f)
     obj2.fieldWork()
}
```

Output:

```
Name is Ashu.
Salary is 40000.0
Name Ashu of department Development with salary 40000.0.
Programming is my passion.

Name is Ajay.
Salary is 30000.0
Name Ajay of department Marketing with salary 30000.0.
Travelling is my hobby.
```

When an object of derived class is created, it calls its superclass first and executes init block of base class followed by its own.

Kotlin Inheritance and secondary constructor

If derived class does not contain any primary constructor then it is required to call the base class secondary constructor from derived class using super keyword.

For example,

```
open class Patent {
     constructor(name: String, id: Int) {
          println("execute super constructor $name: $id")
     }
}

class Child: Patent {
     constructor(name: String, id: Int, dept: String):
          super(name, id) {
          print("execute child class constructor with
                    property $name, $id, $dept")
     }
}
```

```
    fun main(args: Array<String>) {
        val child = Child("Ashu",101, "Developer")
    }
```

Output:

```
execute super constructor Ashu: 101
execute child class constructor with property Ashu, 101, Developer
```

Kotlin Method Overriding

Method overriding means providing the specific implementation of method of super (parent) class into its subclass (child) class.

In other words, when subclass redefines or modifies the method of its superclass into subclass, it is known as method overriding.

Method overriding is only possible in inheritance.

Rules of method overriding

- Parent class and its method or property which is to be overridden **must be open** (non-final).
- Method name of base class and derived class must have same.
- Method must have same parameter as in base class.

Example of inheritance without overriding

```
open class Bird {
    open fun fly(){
        println("Bird is flying...")
    }
}
class Parrot: Bird() {
}
class Duck: Bird() {
}
fun main(args: Array<String>) {
    val p = Parrot()
    p.fly()
    val d = Duck()
    d.fly()
}
```

:Output:

```
Bird is flying...
Bird is flying...
```

In above example, a program without overriding the method of base class we found that both derived classes Parrot and Duck perform the same common operation.

To overcome with this problem we use the concept of method overriding.

Example of Kotlin method overriding

```
open class Bird {
    open fun fly() {
          println("Bird is flying...")
    }
}
class Parrot: Bird() {
    override fun fly() {
          println("Parrot is flying...")
    }
}
class Duck: Bird() {
    override fun fly() {
          println("Duck is flying...")
    }
}
fun main(args: Array<String>) {
     val p = Parrot()
     p.fly()
     val d = Duck()
     d.fly()
}
```

:Output:

```
Parrot is flying...
Duck is flying...
```

Example of Kotlin property overriding

```
open class Bird {
    open var color = "Black"
    open fun fly() {
          println("Bird is flying...")
    }
}
class Parrot: Bird() {
    override var color = "Green"
    override fun fly() {
          println("Parrot is flying...")
    }
}
class Duck: Bird() {
    override var color = "White"
    override fun fly() {
          println("Duck is flying...")
    }
}
fun main(args: Array<String>) {
     val p = Parrot()
     p.fly()
     println(p.color)
     val d = Duck()
     d.fly()
     println(d.color)
}
```

Output:

```
Parrot is flying...
Green
Duck is flying...
White
```

Kotlin superclass implementation

Derived class can also call its superclass methods and property using **super** keyword.

For example:

```
open class Bird {
    open var color = "Black"
    open fun fly() {
          println("Bird is flying...")
    }
}
class Parrot: Bird() {
     override var color = "Green"
     override fun fly() {
          super.fly()
          println("Parrot is flying...")
    }
}

fun main(args: Array<String>) {
     val p = Parrot()
     p.fly()
     println(p.color)
}
```

Output:

```
Bird is flying...
Parrot is flying...
Green
```

Kotlin multiple class implementation

In Kotlin, derived class uses a supertype name in angle brackets, e.g. super<Base> when it implements same function name provided in multiple classes.

For example, a derived class **Parrot** extends its superclass **Bird** and implement Duck interface containing same function **fly()**.

To call particular method of each class and interface we must be mention supertype name in angle brackets as super<Bird>.fly() and super<Duck>.fly() for each method.

```
open class Bird {
    open var color = "Black"
    open fun fly() {
          println("Bird is flying...")
    }
}
interface Duck {
     fun fly() {
          println("Duck is flying...")
     }
}
class Parrot: Bird(),Duck {
    override var color = "Green"
    override fun fly() {
          super<Bird>.fly()
          super<Duck>.fly()
          println("Parrot is flying...")
    }
}
fun main(args: Array<String>) {
     val p = Parrot()
     p.fly()
     println(p.color)
}
```

Output:

```
Bird is flying...
Duck is flying...
Parrot is flying...
Green
```

Kotlin Abstract class

- A class which is declared with **abstract** keyword is known as abstract class.
- An abstract class cannot be instantiated.

  Means, we cannot create object of abstract class.
- The method and properties of abstract class are non-abstract unless they are explicitly declared as abstract.

Declaration of abstract class

```
abstract class A {
var x = 0
    abstract fun doSomething()
}
```

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.
- ✓ Abstract class or abstract function does not need to annotate with open keyword as they are open by default.
- ✓ Abstract member function does not contain its body.
- ✓ The member function cannot be declared as abstract if it contains in body in abstract class.

Example of abstract class that has abstract method

In this example, there is an abstract class Car that contains an abstract function run(). The implementation of run() function is provided by its subclass Honda.

```
abstract class Car{
    abstract fun run()
}
class Honda: Car(){
   override fun run(){
          println("Honda is running safely..")
   }
}
fun main(args: Array<String>){
     val obj = Honda()
     obj.run();
}
```

A **non-abstract** open member function can be over ridden in an abstract class.

```
open class Car {
    open fun run() {
          println("Car is running..")
    }
}
abstract class Honda : Car() {
    override abstract fun run()
}
class City: Honda(){
    override fun run() {
      //  TODO("not implemented") //To change body of created
          functions use File | Settings | File Templates.
          println("Honda City is running..")
    }
}
fun main(args: Array<String>){
     val car = Car()
     car.run()
     val city = City()
     city.run()
}
```

Output:

```
Car is running..
Honda City is running..
```

- In above example, An abstract class Honda extends the class Car and its function **run()**.
- Honda class override the **run()** function of Car class.
- The Honda class did not give the implementation of **run()** function as it is also declared as abstract.
- The implementation of abstract function **run()** of Honda class is provided by City class.

Example of real scenario of abstract class

In this example, an abstract class Bank that contains an abstract function **simpleInterest()** accepts three parameters p,r,and t.

The class SBI and PNB provides the implementation of **simpleInterest()** function and returns the result.

```
abstract class Bank {
 abstract fun simpleInterest(p: Int, r: Double, t: Int): Double
}

class SBI : Bank() {
    override fun simpleInterest(p: Int, r: Double, t: Int):
     Double{
        return (p*r*t)/100
    }
}
class PNB : Bank() {
    override fun simpleInterest(p: Int, r: Double, t: Int):
     Double{
        return (p*r*t)/100
    }
}
fun main(args: Array<String>) {
     var sbi: Bank = SBI()
     val sbiint = sbi.simpleInterest(1000,5.0,3)
     println("SBI interest is $sbiint")
     var pnb: Bank = PNB()
     val pnbint = pnb.simpleInterest(1000,4.5,3)
     println("PNB interest is $pnbint")
}
```

Output:

```
SBI interest is 150.0
PNB interest is 135.0
```

Kotlin Interface

- An interface is a blueprint of class.
- Kotlin interface is similar to Java 8.
- It contains abstract method declarations as well as implementation of method.

Defining Interface

An interface is defined using the keyword **interface**. For example:

```
interface MyInterface {
      val id: Int // abstract property
      fun absMethod()// abstract method
      fun doSomthing() {
       // optional body
      }
}
```

The methods which are only declared without their method body are **abstract** by default.

Why use Kotlin interface?

Following are the reasons to use interface:

- Using interface supports functionality of multiple inheritance.
- It can be used achieve to loose coupling.
- It is used to achieve abstraction.

Subclass extends only one super class but implements multiple interfaces. Extension of parent class or interface implementation are done using (:) operator in their subclass.

Implementing Interfaces

In this example, we are implementing the interface MyInterface in InterfaceImp class.

InterfaceImp class provides the implementation of property id and abstract method absMethod() declared in MyInterface interface.

```
interface MyInterface  {
     var id: Int             // abstract property
     fun absMethod():String    // abstract method
     fun doSomthing() {
          println("MyInterface doing some work")
    }
}
class InterfaceImp : MyInterface {
    override var id: Int = 101
    override fun absMethod(): String{
               return "Implementing abstract method.."
    }
}
fun main(args: Array<String>) {
     val obj = InterfaceImp()
     println("Calling overriding id value = ${obj.id}")
     obj.doSomthing()
     println(obj.absMethod())
}
```

Output:

```
Calling overriding id value = 101
MyInterface doing some work
Implementing abstract method..
```

Implementing multiple interface

We can implement multiple abstract methods of different interfaces in same class.

All the abstract methods must be implemented in subclass.

The other non-abstract methods of interface can be called from derived class.

For example, creating two interface *MyInterface1* and *MyInterface2* with abstract methods *doSomthing()* and *absMethod()* respectively.

These abstract methods are overridden in derive class *MyClass*.

```
interface MyInterface1 {
    fun doSomthing()
}
interface MyInterface2 {
    fun absMethod()
}
class MyClass : MyInterface1, MyInterface2 {
    override fun doSomthing() {
          println("overriding doSomthing() of MyInterface1")
    }
    override fun absMethod() {
          println("overriding absMethod() of MyInterface2")
    }
}
fun main(args: Array<String>) {
     val myClass = MyClass()
     myClass.doSomthing()
     myClass.absMethod()
}
```

Output:

```
overriding doSomthing() of MyInterface1
overriding absMethod() of MyInterface2
```

Resolving different Interfaces having same method overriding conflicts

Let's see an example in which interface MyInterface1 and interface MyInterface2 both contains same non-abstract method.

A class MyClass provides the implementation of these interfaces.

Calling the method of interface using object of MyClass generates an error.

```
interface MyInterface1 {
    fun doSomthing(){
          println("overriding doSomthing() of MyInterface1")
    }
}
interface MyInterface2 {
    fun doSomthing(){
          println("overriding doSomthing() of MyInterface2")
    }
}
class MyClass : MyInterface1, MyInterface2 {

}
fun main(args: Array<String>) {
     val myClass = MyClass()
     myClass.doSomthing()
}
```

Output:

```
Kotlin: Class 'MyClass' must override public open fun doSomthing():
Unit defined in MyInterface1 because it
inherits multiple interface methods of it
```

To solve the above problem we need to specify particular method of interface which we are calling. Let's see an example below.

In below example, two interfaces MyInterface1 and MyInterface2 contain two abstract methods adsMethod() and absMethod(name: String) and non-abstract method doSomthing() in both respectively.

A class **MyClass** implements both interface and override abstract method **absMethod()** and **absMethod(name: String)**.

To override the non-abstract method doSomthing() we need to specify interface name with method using super keyword as super<interface_name>.methodName().

```
interface MyInterface1 {
    fun doSomthing() {
          println("MyInterface 1 doing some work")
    }
    fun absMethod()
}

interface MyInterface2 {
    fun doSomthing(){
          println("MyInterface 2 doing some work")
    }
    fun absMethod(name: String)
}

class MyClass : MyInterface1, MyInterface2 {
    override fun doSomthing() {
        super<MyInterface2>.doSomthing()
    }
    override fun absMethod() {
          println("Implements absMethod() of MyInterface1")
    }
    override fun absMethod(n: String) {
          println("Implements absMethod(name) of
               MyInterface2 name is  $n")
    }
}
```

```
fun main(args: Array<String>) {
     val myClass = MyClass()
     myClass.doSomthing()
     myClass.absMethod()
     myClass.absMethod("Ashu")
}
```

Output:

```
MyInterface 2 doing some work
Implements absMethod() of MyInterface1
Implements absMethod(name) of MyInterface2 name is
Ashu
```

---------- **Example: 02** ----------

```
interface MyInterface1 {
    fun doSomthing() {
          println("MyInterface 1 doing some work")
    }
    fun absMethod()
}

interface MyInterface2 {
    fun doSomthing() {
          println("MyInterface 2 doing some work")
    }
   fun absMethod() {
          println("MyInterface 2 absMethod")
    }
}

class C : MyInterface1 {
    override fun absMethod() {
          println("MyInterface1 absMethod implementation")
    }
}

class D : MyInterface1, MyInterface2 {
    override fun doSomthing() {
        super<MyInterface1>.doSomthing()
        super<MyInterface2>.doSomthing()
    }
    override fun absMethod() {

        super<MyInterface2>.absMethod()
    }
}
```

```
fun main(args: Array<String>) {
     val d = D()
     val c = C()
     d.doSomthing()
     d.absMethod()
     c.doSomthing()
     c.absMethod()
}
```

:Output:

```
MyInterface 1 doing some work
MyInterface 2 doing some work
   MyInterface 2 absMethod
MyInterface 1 doing some work
MyInterface1 absMethod implementation
```