8.1 Kotlin Generics:

Generics are the powerful features that allow to define classes, methods, and properties etc. which can be accessed using different types. The type differences of classes, methods, etc. are **checked at compile-time**.

The generic type class or method is declared as parameterized type.

A parameterized type is an instance of generic type with actual type arguments.

The parameterized types are declared using angle brackets **<>.**

Generics are mostly used in collections.

Advantage of Generics

Following are the key advantages of using generics:

- Type-safety: Generic allows to hold only single type of object.
- Generic does not allow to store other object.
- Type casting is not required: There is no need to typecast the object.
- Compile time checking: Generics code is checked at compile time so that it can avoid any problems at runtime

Let's see a problem without using the generics.

In this example, we create a **Person** class with primary constructor having single parameter.

Now, we want to pass the different type of data in object of Person class (say Int type as Person(30) and String type as Person("40")).

The primary constructor of Person class accept Int type Person(30) and regrets String type Person("40").

It generates a compile time error as type mismatch.

```
class Person (age:Int){
     var age: Int = age
     init{
          this.age= age
          println(age)
     }
}
fun main(args: Array<String>){
     var ageInt: Person = Person(30)
     var ageString: Person = Person("30")// compile time error
}
```

To solve the above problem, we use a generic type class which is a user defined class that accepts different type of parametersin single class.

Let's rewrite the above code using generic type.

A class Person of type **<T>** is a general type class that accepts both **Int** and **String** types of parameter.

In other words, the type parameter **<T>** is a place holder that will be replaced by type argument.

It will be replaced when the generic type is instantiated.

```
     class Person<T>(age: T){
          var age: T = age
          init{
               this.age= age
               println(age)
          }
     }
     fun main(args: Array<String>){
          var ageInt: Person<Int> = Person<Int>(30)
          var ageString: Person<String> = Person<String>("40")
     }
```

In above example, when the object of Person class is created using type Int as Person<Int>(30) and Person<String>("40"), it replaces the Person class of type **T** with Int and String respectively.

Kotlin generic example

In this example, we are accessing the generic method of collection type (ArrayList)

```
fun main(args: Array<String>){
     val stringList: ArrayList<String> =
                          arrayListOf<String>("Ashu","Ajay")
     val s: String = stringList[0]
     println("printing the string value of stringList: $s")
     printValue(stringList)
     val floatList: ArrayList<Float> =
                          arrayListOf<Float>(10.5f,5.0f,25.5f)
     printValue(floatList)
}

fun <T>printValue(list: ArrayList<T>){
    for(element in list){
          println(element)
    }
}
```

Output:

```
printing the string value of
stringList: Ashu
Ashu
Ajay
10.5
5.0
25.5
```

Kotlin generic extension function example

As extension function allows to add methods to class without inherit a class or any design pattern.

```
fun main(args: Array<String>){
     val stringList: ArrayList<String> =
                          arrayListOf<String>("Ashu","Ajay")
     stringList.printValue()
     val floatList: ArrayList<Float> =
                          arrayListOf<Float>(10.5f,5.0f,25.5f)
     floatList.printValue()
}
fun <T>ArrayList<T>.printValue(){
    for(element in this){
          println(element)
    }
}
```

8.2 Kotlin Data class

Data class is a simple class which is used to hold data/state and contains standard functionality.

A ***data*** keyword is used to declare a class as a data class.

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

Declaring a ***data*** class must contains at least *one primary constructor* with property argument (**val** or **var**).

Data class internally contains the following functions:

- equals(): Boolean
- hashCode(): Int
- toString(): String
- component() functions corresponding to the properties
- copy()

➢ Due to presence of above functions internally in data class, the data class eliminates the boilerplate code.

A compression between Java data class and Kotlin data class

https://www.javatpoint.com/kotlin-data-class

Requirements of data class

In order to create a data class, we need to fulfill the following requirements:

- Contain primary constructor with at least one parameter.
- Parameters of primary constructor marked as **val** or **var**.
- Data class cannot be *abstract*, *inner*, *open* or *sealed*.
- Before 1.1, data class may only implements interface. After that data classes may extend other classes.

8.3 Kotlin Sealed Class

- Sealed class is a class which restricts the class hierarchy.
- A class can be declared as sealed class using "***sealed***" keyword before the class name.
- It is used to represent restricted class hierarchy.
- Sealed class is used when the object have one of the types from limited set, but cannot have any other type.
- The constructors of sealed classes are private in default and cannot be allowed as **non-private**.

https://www.javatpoint.com/kotlin-sealed-class