Android Development with Kotlin

Generics in Kotlin

Type and Subtype

Type of an object is the class of which the object is an instance of where as a subtype is the inherited class, or subclass of a type class. In Kotlin, every user-defined class generates two types with the first one being a nullable type and the second one being a non-nullable type. The non-nullable type is the subset of a nullable type.
An object of a type can be initialized/assigned with an instance of its subtype but the reverse isn't allowed. **For ex**, a nullable string object can be initialized with a non-nullable string but the reverse is restricted and will result in an error.

```
var string1: String = "abc"
var string2: String? = string1 //this is a valid statement
var string3: String = string2  //but this is not
```

Generic Functions, Classes and Constraints

A generic function is a function that can be used for a number of type classes. Generic functions specify a type parameter during their declaration and any statement calling these functions, has to specify the type parameter while invoking the said function. Similarly, a generic class is a class that specifies a type parameter during its declaration and can be used with different types of objects by merely changing the type parameter.
Generic functions and classes allow code reusability and allow all us to use these functions for any classes that inherit the class "Any?". Although in some cases, the generic property of a class needs to be restricted. For example, if a generic class is designed to provide arithmetic operations, a String type should not be allowed as a type parameter. This can be restricted by the use of ":" operator for single constraints and the "where operator for multiple constraints.

```
fun<T: Any> function_name
//the type is restricted to non-nullable type only
fun<T> function_name where T:Number, T:Serializable
//multiple constraints
```

Invariance

Polymorphism is one of the major properties offered by an Object Oriented Programming Language but in case of generics the complex classes i.e. the generic classes do not follow the polymorphic properties of their type classes. In other words, a type class in its complex form cannot be assigned an object of its subclass in the same complex formation.

```
var generic:Generic<Type> = Generic<Sub-Type>()
//is an invalid statement
```

This inability of the complex classes to follow polymorphism is known as Invariance.

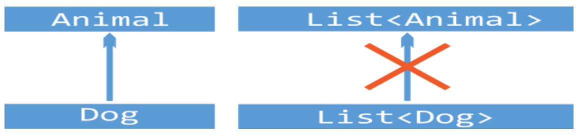

Image src: https://miro.medium.com/max/1400/0*CFQD7UM9u55d9xPl.

Covariance

Covariance is a property that allows a complex generic classes to be assigned the instances of their sub-classes. This is possible by the use of the **“out”** keyword before the type parameter. The only condition is that, the generic class with covariance can only be a producer of its type and not a consumer. In other words, the class can only have functions that return the type parameter but cannot have functions that take type as its input.

```
val dogList: List<Dog> = listOf(Dog(10), Dog(20))
val animalList: List<Animal> = dogList
```

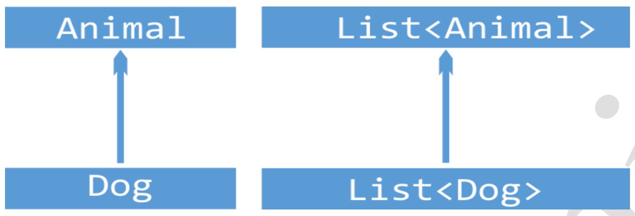

image src: https://miro.medium.com/max/1400/0*FGJq5lPte9p7JINm.

Contravariance

Contravariance on the other hand, allows a complex generic subclass to be assigned the instances of their parent classes. This is possible by the use of the “in” keyword before the type parameter. The only condition is that, the generic class with contravariance can only be a consumer of its type and not a producer. In other words, the class can only have functions that possess the type parameter as its arguments but cannot have functions that return type as its output. a

```
interface Compare<in T> {// specified using in keyword
fun compare(first: T, second: T): Int
}
val animalCompare: Compare<Animal> = object: Compare<Animal> {
  override fun compare(first: Animal, second: Animal): Int {
    return first.size - second.size
  }
}
val spiderCompare: Compare<Spider> = animalCompare // Works nicely!
```

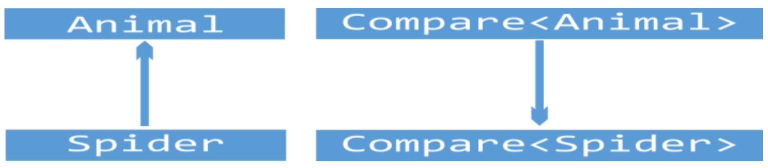

image src: https://miro.medium.com/max/1400/0*ORGz8lT_rHFwmbzH..

Type Projections and Star Projections

Variance in Kotlin can be achieved in two ways, declaration-site and use-site. Covariance and Contravariance are the examples of declaration-site variance and type projections are an example of use-site variance.
Type projections define the variance of the type parameter when it is used. In practice this means that it is defined separately for each method using bounded wildcards.
Star Projections on the other hand are used to specify that the type of the generic function is irrelevant to the output of the program and is inconsequential throughout the code segment.

```
fun acceptStarArray(array: Array<*>) {}

val StringArray = arrayOf("Hello", "World")
acceptAnyArray(StringArray)
acceptStarArray(StringArray)
```

Reified Types and Inline Functions

One of the limitations that most frustrates Java developers when using generics is not being able to use the type directly. Normally this is solved by passing the class as a function parameter, making the code more complex and unattractive. This can be solved with the use of inline function in combination with the reified types.
The "inline" keyword when used before a function declaration informs the compiler to replace the function call with the statements of the function itself, this in turn reduces the run time of the program as the time used for function calls is eliminated but the memory overhead of the program increases significantly. A "reified" keyword can only be used with an inline function. When used, this keyword allows the function to access the type specified in its type parameter.

First Approach: Without reified

```
fun <T> String.toKotlinObject(): T {
      val mapper = jacksonObjectMapper()          //does not compile!
      return mapper.readValue(this, T::class.java)
}
```

Second Approach: With reified

```
inline fun <reified T: Any> String.toKotlinObject(): T {
    val mapper = jacksonObjectMapper()
    return mapper.readValue(this, T::class.java)
}
```