9.1 Delegation:

Kotlin supports "delegation" design pattern by introducing a new keyword "**by**".

Using this keyword or delegation methodology, Kotlin allows the derived class to access all the implemented public methods of an interface through a specific object.

The following example demonstrates how this happens in Kotlin.

https://www.tutorialspoint.com/kotlin/kotlin_delegation.htm

```
interface Base{
   fun printMe() //abstract method
}

class BaseImpl(val x: Int) : Base {
     override fun printMe() {
          println(x)
     }   //implementation of the method
}

class Derived(b: Base) : Base by b
          // delegating the public method on the object b

fun main(args: Array<String>){
     val b = BaseImpl(10)
     Derived(b).printMe() // prints 10 :: accessing the
     printMe() method
}
```

Property Delegation

Delegation means passing the responsibility to another class or method.

When a property is already declared in some places, then we should reuse the same code to initialize them.

In the following examples, we will use some standard delegation methodology provided by Kotlin and some standard library function while implementing delegation in our examples.

Using Lazy()

Lazy is a lambda function which takes a property as an input and in return gives an instance of Lazy<T>, where <T> is basically the type of the properties it is using.

Let us take a look at the following to understand how it works.

```
val myVar: String by lazy {
   "Hello"
}
fun main(args: Array<String>) {
   println(myVar +" My dear friend")
}
```

https://www.tutorialspoint.com/kotlin/kotlin_delegation.htm

Delegetion.Observable()

Observable() takes *two arguments* to initialize the object and returns the same to the called function.

In the following example, we will see how to use Observable() method in order to implement delegation.

```
import kotlin.properties.Delegates

class User {
   var name: String by Delegates.observable(
                     "Welcome to Tutorialspoint.com") {
      prop, old, new ->
      println("$old -> $new")
   }
}
fun main(args: Array<String>) {
   val user = User()
   user.name = "first"
   user.name = "second"
}
```

---

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

```
interface Base {
    fun printMessage()
    fun printMessageLine()
}

class BaseImpl(val x: Int) : Base {
    override fun printMessage() { print(x) }
    override fun printMessageLine() { println(x) }
}

class Derived(b: Base) : Base by b {
    override fun printMessage() { println("abc") }
}

fun main() {
    val b = BaseImpl(10)
    Derived(b).printMessage()
    Derived(b).printMessageLine()
}
```

Output:

```
abc
10
```

```
interface Base {
    val message: String
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override val message = "BaseImpl: x = $x"
    override fun print() { println(message) }
}

class Derived(b: Base) : Base by b {
    // This property is not accessed from b's implementation of `print`
    override val message = "Message of Derived"
}

fun main() {
    val b = BaseImpl(10)
    val derived = Derived(b)
    derived.print()
    println(derived.message)
}
```

Output:

```
BaseImpl: x = 10
Message of Derived
```

Delegated Properties

https://kotlinlang.org/docs/reference/delegated-properties.html

There are certain common kinds of properties, that, though we can implement them manually every time we need them, would be very nice to implement once and for all, and put into a library.

Examples include:

lazy properties: the value gets computed only upon first access;

observable properties: listeners get notified about changes to this property;
storing properties in a **map**, instead of a separate field for each property.

To cover these (and other) cases, Kotlin supports ***delegated properties:***

```
class Example {
    var p: String by Delegate()
}
```

The syntax is: **val/var** <property name>: <Type> by <expression>.

The expression after **by** is the delegate, because **get()** (and **set()**) corresponding to the property will be delegated to its getValue() and setValue() methods.

Property delegates don't have to implement any interface, but they have to provide a **getValue()** function (and **setValue()** – for **vars**).

```
class Delegate {
      operator fun getValue(thisRef: Any?, property:
KProperty<*>): String {
         return "$thisRef, thank you for delegating '$
{property.name}' to me!"
    }

      operator fun setValue(thisRef: Any?, property:
KProperty<*>, value: String) {
            println("$value has been assigned to '$
{property.name}' in $thisRef.")
    }
}
```

Standard Delegates

The Kotlin standard library provides factory methods for several useful kinds of delegates.

- lazy
- observable

https://kotlinlang.org/docs/reference/delegated-properties.html

Storing Properties in a Map

```
-----------------------------------------
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}

fun main(){
    val user = User(mapOf(
            "name" to "John Doe",
            "age"  to 25
        ))
/**-------- Compilation Error -------------------
user.name = "JAINUL"   <<error: val cannot be reassigned
user.age = 44          <<error: val cannot be reassigned
-----------------------------------------*/
    println(user.name) // Prints "John Doe"
    println(user.age)  // Prints 25
}
```

```
class User(val map: MutableMap<String, Any>) {
    var name: String by map
    var age: Int     by map
}

fun main(){
    val user = User(mutableMapOf(
                          "name" to "John Doe",
                          "age"  to 25
))

    user.name = "JAINUL"
    user.age = 44
    println(user.name) // Prints "JAINUL"
    println(user.age)  // Prints 44
}
```