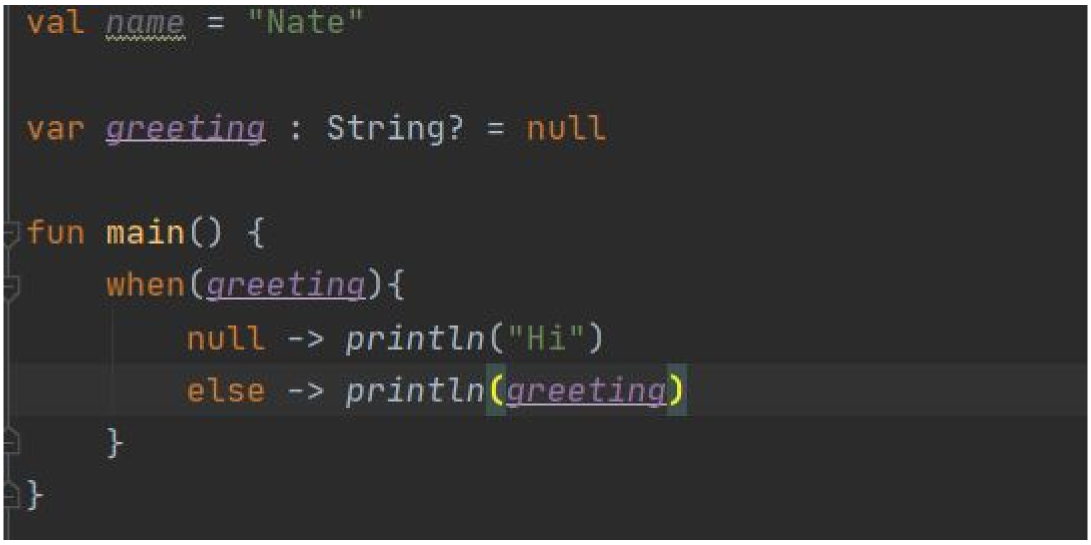
1. 2 Types of Variables : Mutable (value can be changed) and Non-mutable (Local Read Only variables)
Mutable Variable : var   Non-mutable Variable : val

2. Difference between JAVA and Kotlin : types in Kotlin are non null i.e. there is a distinct difference between Nullable String and non-nullable String.

```
val full_name:String? = null
val name:String = null
```

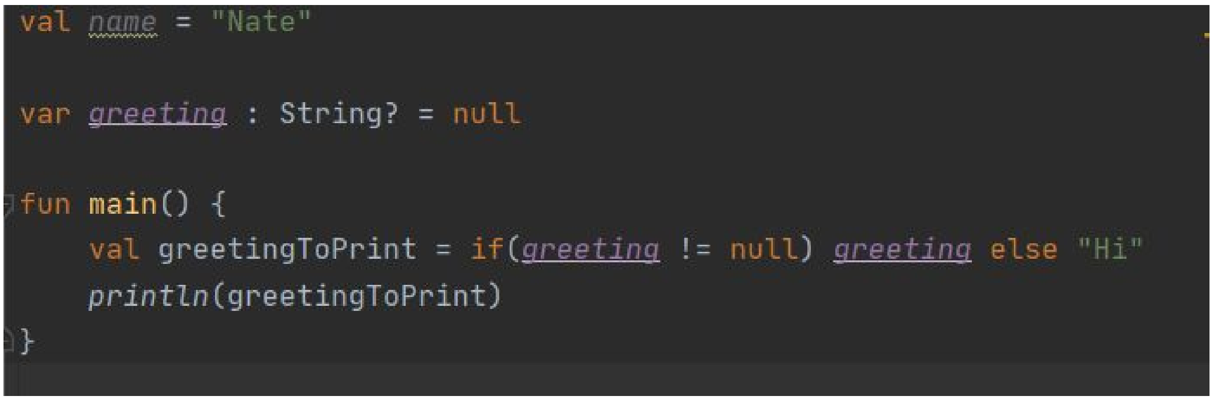
3. If () {} - Same as JAVA

4. When statement is same as switch in JAVA



```
val name = "Nate"

var greeting : String? = null

fun main() {
    when(greeting){
        null -> println("Hi")
        else -> println(greeting)
    }
}
```

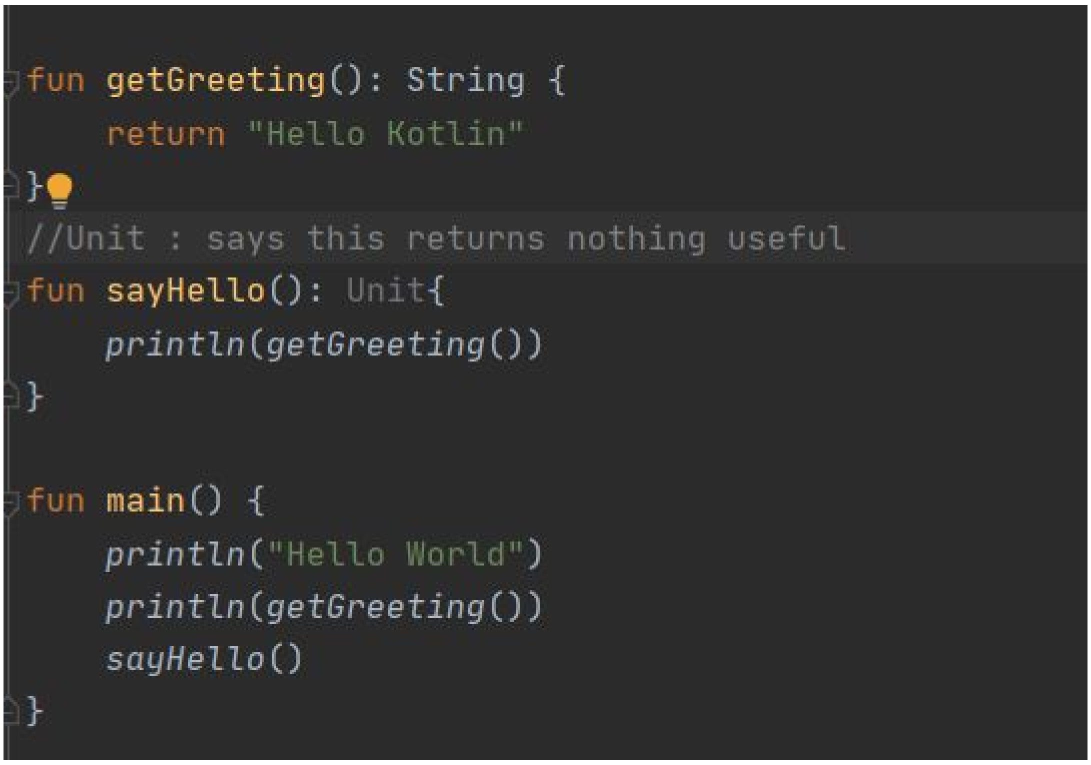
5. If can used to assign values



```
val name = "Nate"

var greeting : String? = null

fun main() {
    val greetingToPrint = if(greeting != null) greeting else "Hi"
    println(greetingToPrint)
}
```

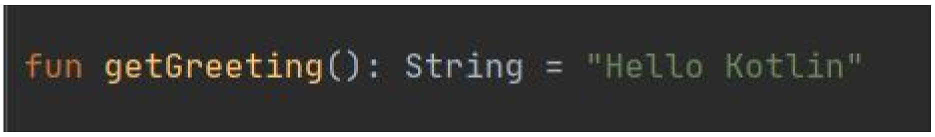
6. Similarly when(...){}can be used

7. Unit : returns Nothing (Functions in Kotlin)



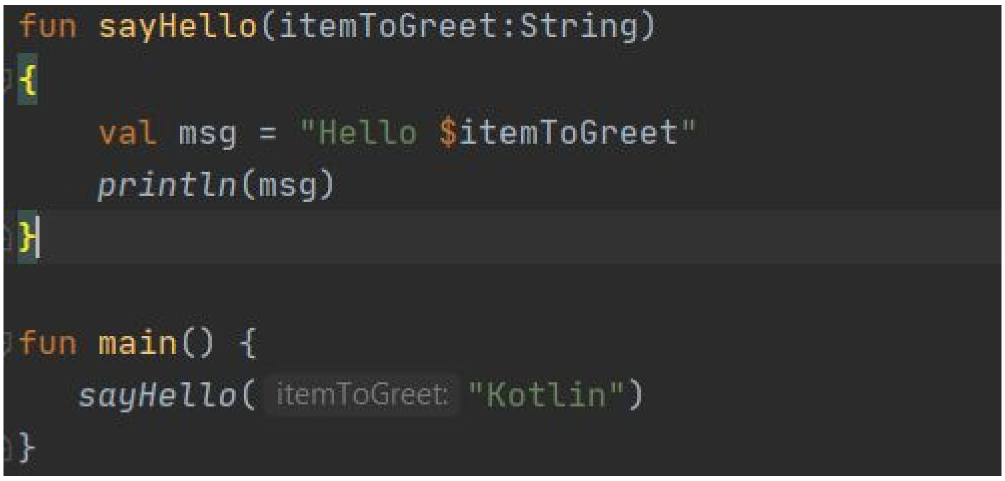
```kotlin
fun getGreeting(): String {
    return "Hello Kotlin"
}
//Unit : says this returns nothing useful
fun sayHello(): Unit{
    println(getGreeting())
}

fun main() {
    println("Hello World")
    println(getGreeting())
    sayHello()
}
```

8. Single Expression Function (Type Inference : So can remove String type)

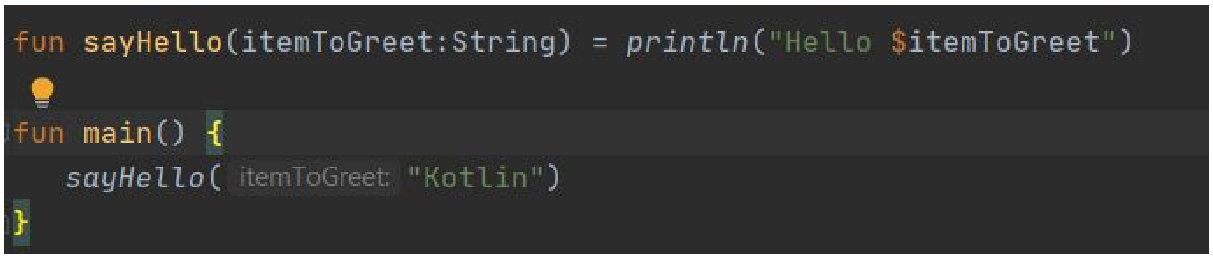


```kotlin
fun getGreeting(): String = "Hello Kotlin"
```

9. String template value in parameter of function

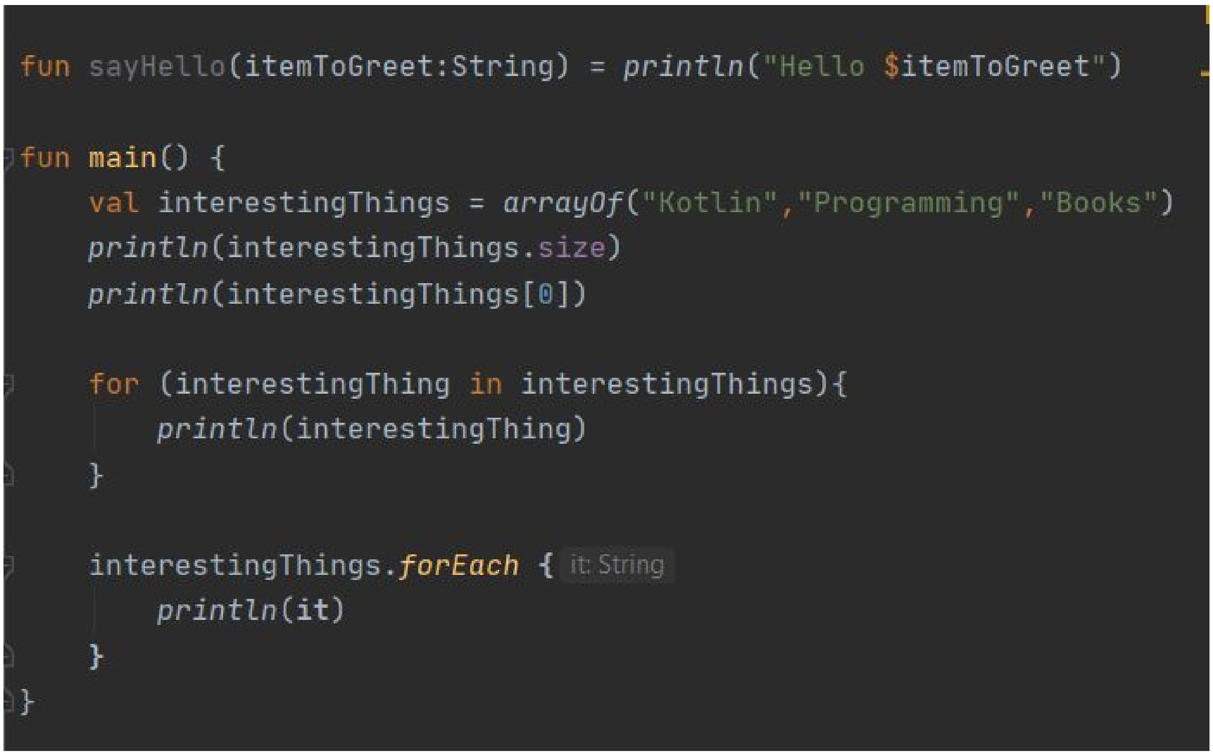


```kotlin
fun sayHello(itemToGreet:String)
{
    val msg = "Hello $itemToGreet"
    println(msg)
}

fun main() {
   sayHello( itemToGreet: "Kotlin")
}
```



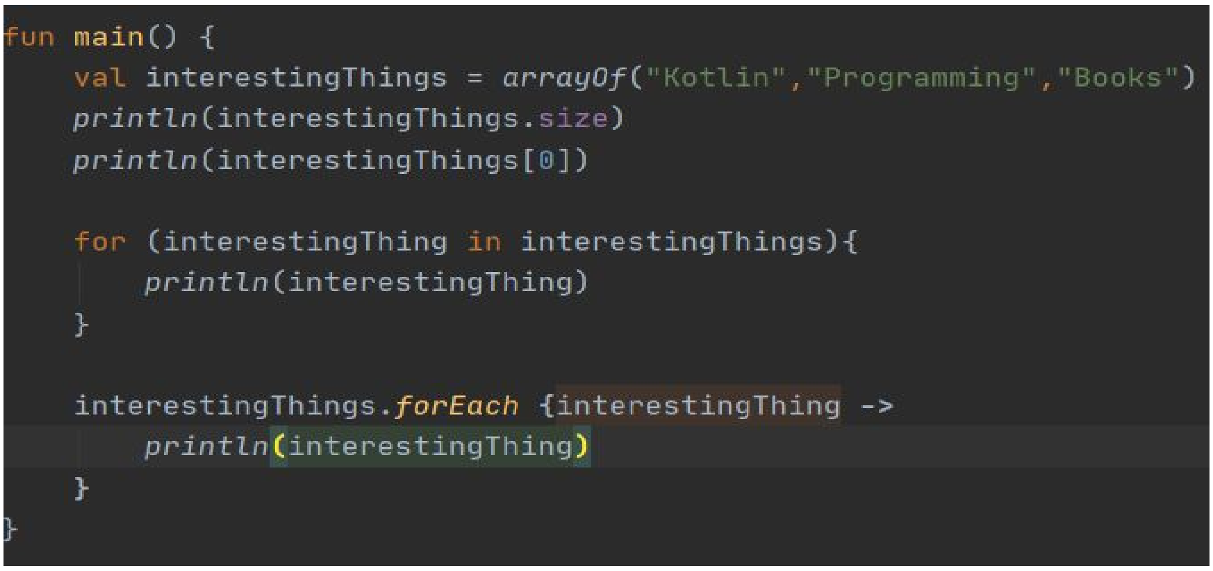
```kotlin
fun sayHello(itemToGreet:String) = println("Hello $itemToGreet")

fun main() {
    sayHello( itemToGreet: "Kotlin")
}
```

10. For loop both JAVA style and functional style



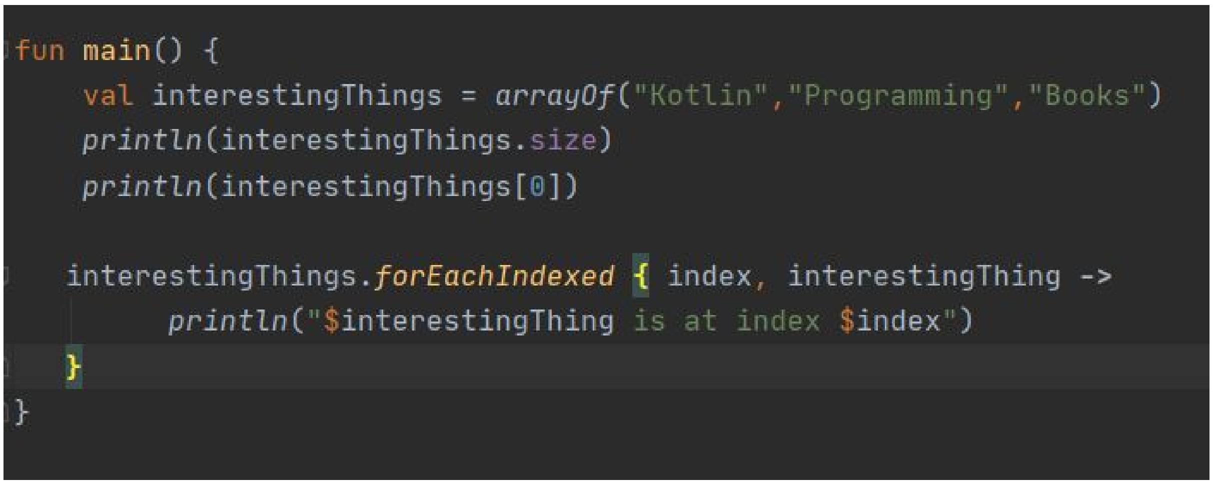
```kotlin
fun sayHello(itemToGreet:String) = println("Hello $itemToGreet")

fun main() {
    val interestingThings = arrayOf("Kotlin","Programming","Books")
    println(interestingThings.size)
    println(interestingThings[0])

    for (interestingThing in interestingThings){
        println(interestingThing)
    }

    interestingThings.forEach { it: String
        println(it)
    }
}
```

11. Lambda Function in Kotlin : Idea of lambda function is that if you have a function and its only parameter is another function then you can omit the parentheses and you can pass that function in specifying these open and close parentheses.



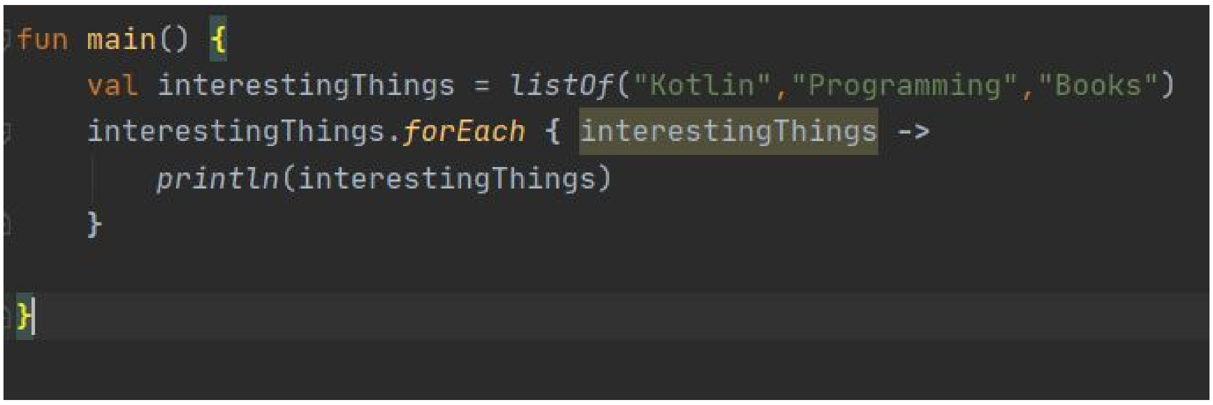
```kotlin
fun main() {
    val interestingThings = arrayOf("Kotlin","Programming","Books")
    println(interestingThings.size)
    println(interestingThings[0])

    for (interestingThing in interestingThings){
        println(interestingThing)
    }

    interestingThings.forEach {interestingThing ->
        println(interestingThing)
    }
}
```

12. Iterate and still maintain the indexes



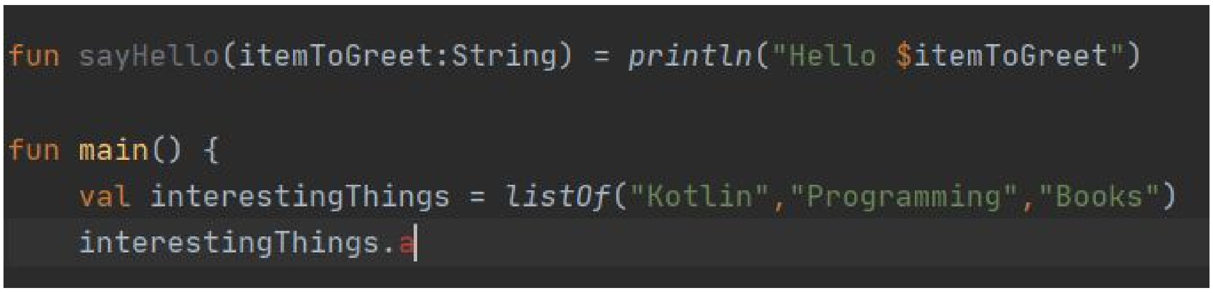
```
fun main() {
    val interestingThings = arrayOf("Kotlin","Programming","Books")
    println(interestingThings.size)
    println(interestingThings[0])

   interestingThings.forEachIndexed { index, interestingThing ->
         println("$interestingThing is at index $index")
   }
}
```

13. Lambda expression with list



```
fun main() {
    val interestingThings = listOf("Kotlin","Programming","Books")
    interestingThings.forEach { interestingThings ->
        println(interestingThings)
    }

}
```

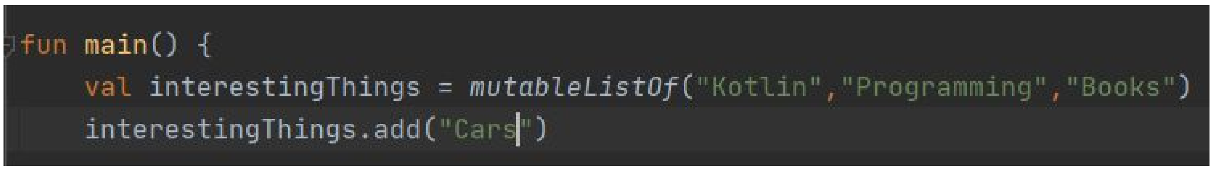
14. Mutable and immutable collections type :
By default a collection type in Kotlin is immutable : once a value is defined for a collections it cannot be modified.



```
fun sayHello(itemToGreet:String) = println("Hello $itemToGreet")

fun main() {
    val interestingThings = listOf("Kotlin","Programming","Books")
    interestingThings.a
```

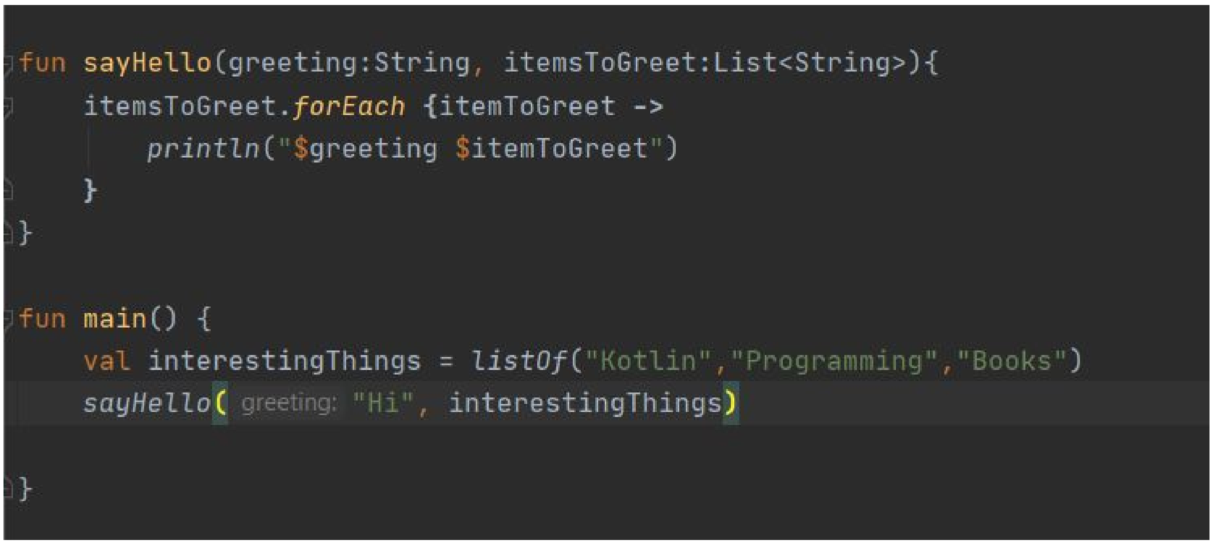
No ADD function

Mutable Collection:



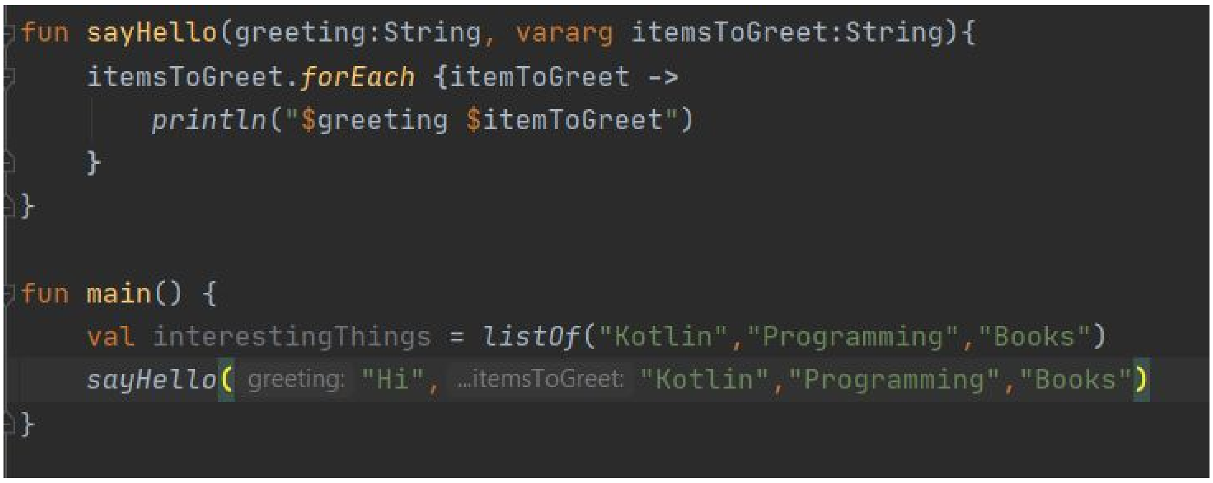
```
fun main() {
    val interestingThings = mutableListOf("Kotlin","Programming","Books")
    interestingThings.add("Cars")
```

15. Functions



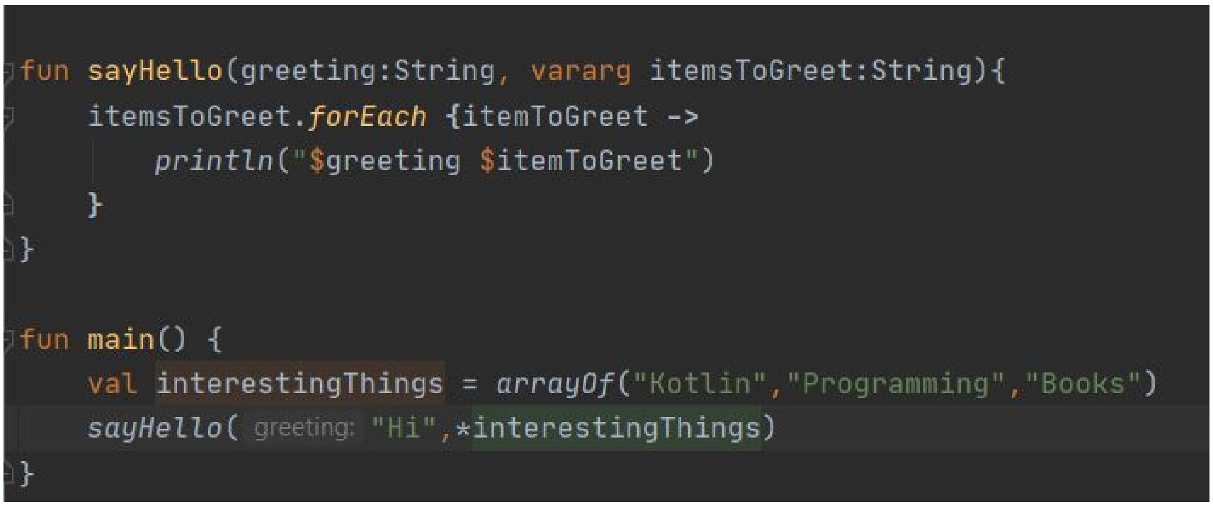
```kotlin
fun sayHello(greeting:String, itemsToGreet:List<String>){
    itemsToGreet.forEach {itemToGreet ->
        println("$greeting $itemToGreet")
    }
}

fun main() {
    val interestingThings = listOf("Kotlin","Programming","Books")
    sayHello( greeting: "Hi", interestingThings)

}
```

16. VarArgs in Kotlin



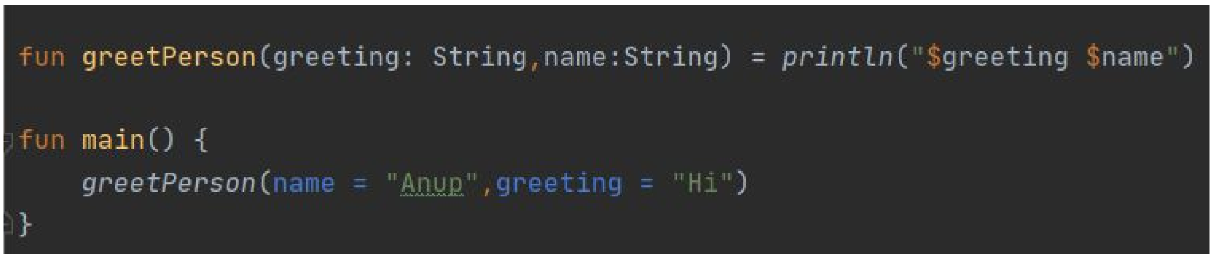
```kotlin
fun sayHello(greeting:String, vararg itemsToGreet:String){
    itemsToGreet.forEach {itemToGreet ->
        println("$greeting $itemToGreet")
    }
}

fun main() {
    val interestingThings = listOf("Kotlin","Programming","Books")
    sayHello( greeting: "Hi", ...itemsToGreet: "Kotlin","Programming","Books")
}
```

17. Existing Collection as a vararg argument requires the spread operator (*) to be used



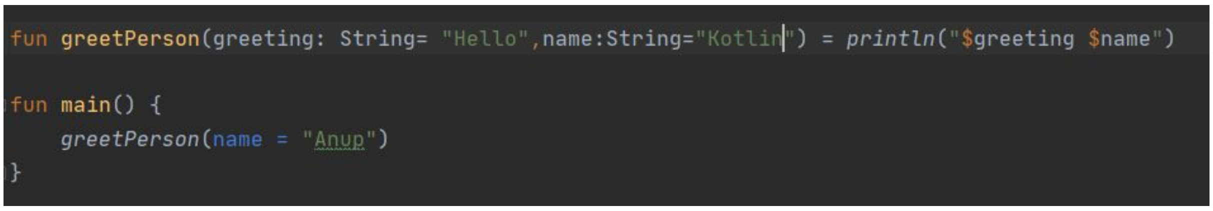
```kotlin
fun sayHello(greeting:String, vararg itemsToGreet:String){
    itemsToGreet.forEach {itemToGreet ->
        println("$greeting $itemToGreet")
    }
}

fun main() {
    val interestingThings = arrayOf("Kotlin","Programming","Books")
    sayHello( greeting: "Hi",*interestingThings)
}
```

18. Named Arguments in Kotlin



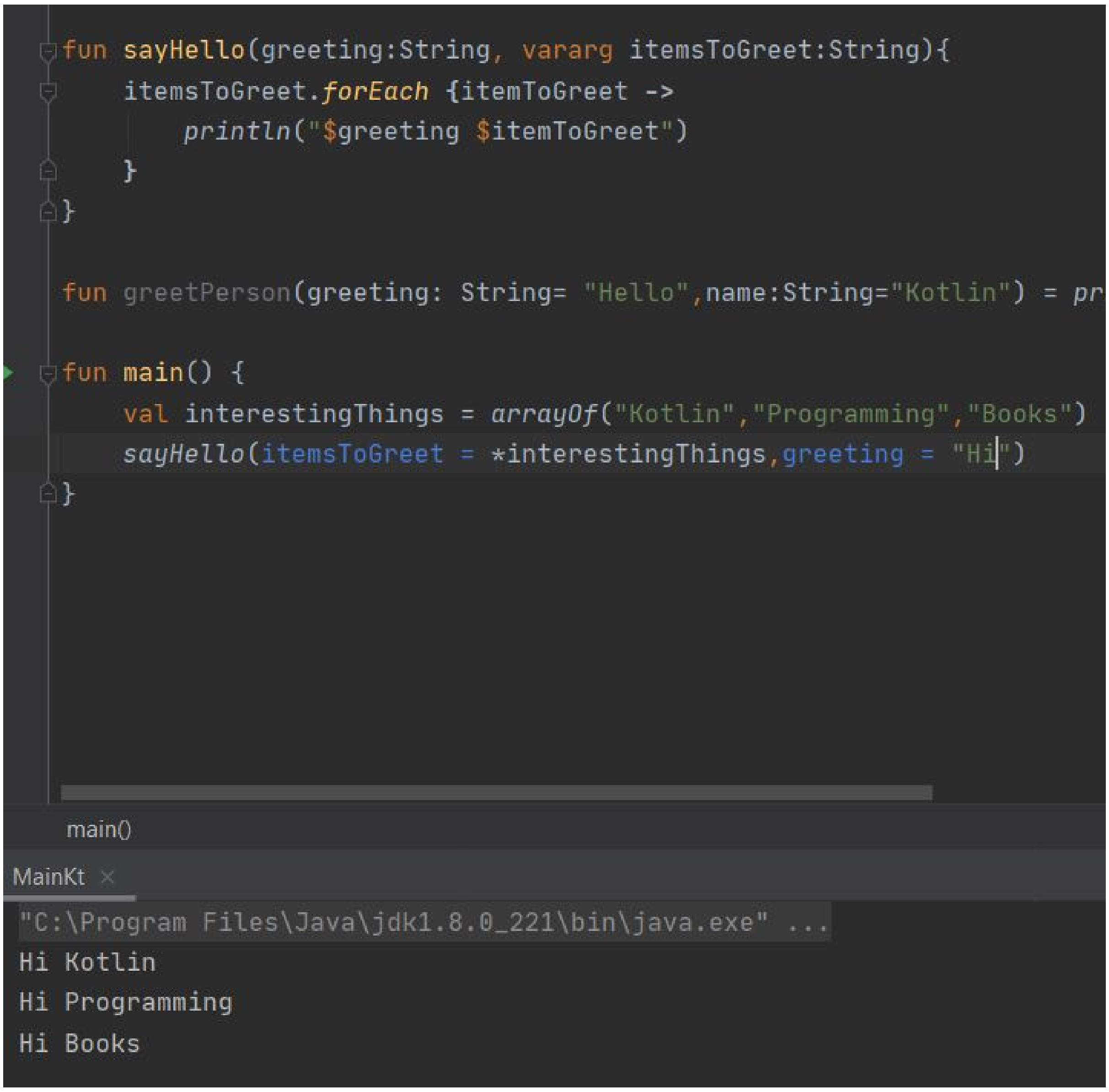
```kotlin
fun greetPerson(greeting: String,name:String) = println("$greeting $name")

fun main() {
    greetPerson(name = "Anup",greeting = "Hi")
}
```

19. Default Value in function



```kotlin
fun greetPerson(greeting: String= "Hello",name:String="Kotlin") = println("$greeting $name")

fun main() {
    greetPerson(name = "Anup")
}
```

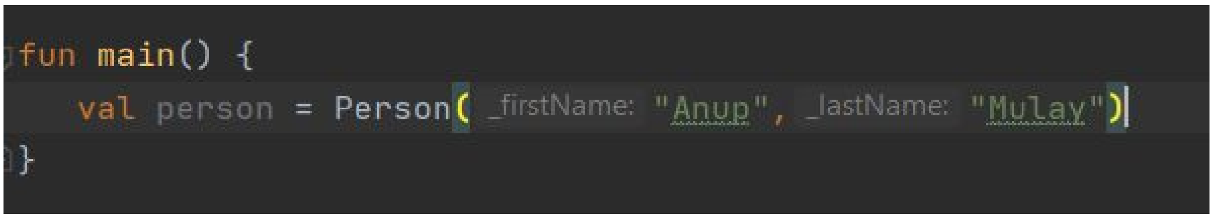
20. As soon as the names argument syntax is used for one argument then everything that follows that must also be named.



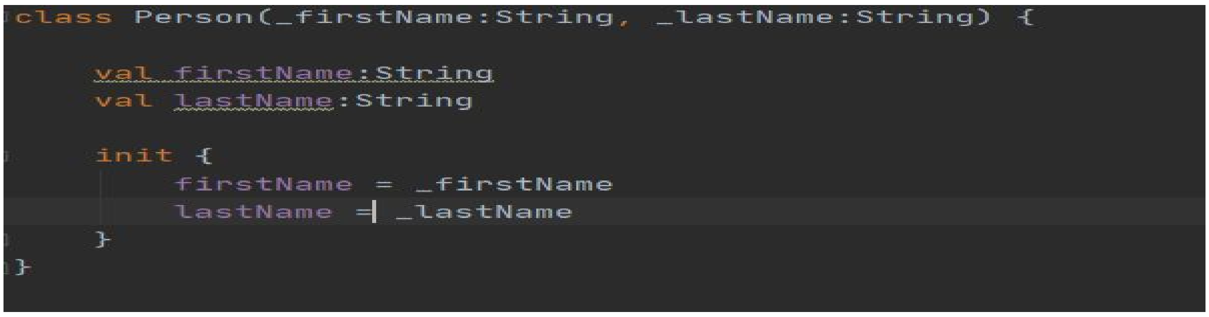
```
fun sayHello(greeting:String, vararg itemsToGreet:String){
    itemsToGreet.forEach {itemToGreet ->
        println("$greeting $itemToGreet")
    }
}

fun greetPerson(greeting: String= "Hello",name:String="Kotlin") = pr

fun main() {
    val interestingThings = arrayOf("Kotlin","Programming","Books")
    sayHello(itemsToGreet = *interestingThings,greeting = "Hi")
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Hi Kotlin
Hi Programming
Hi Books
```

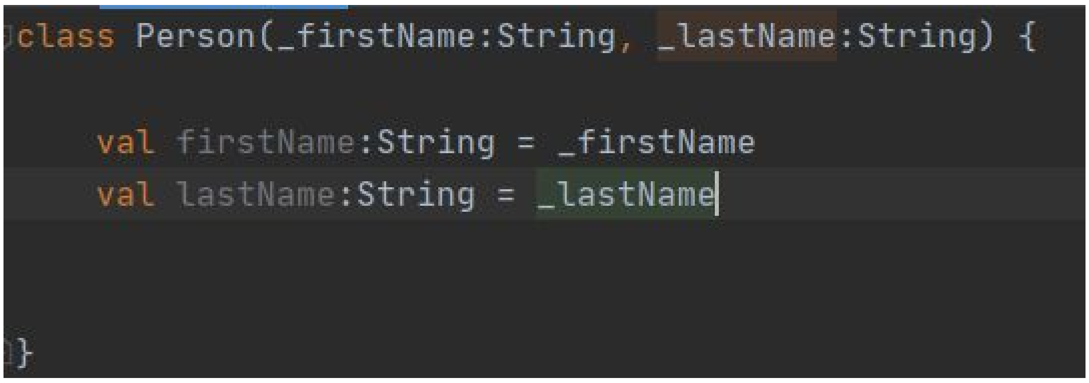
21. Classes in Kotlin



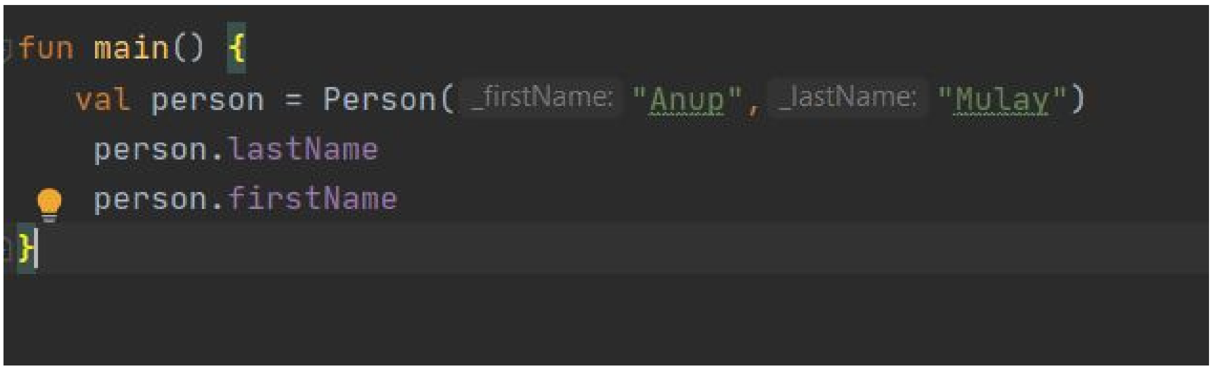
```
fun main() {
    val person = Person( _firstName: "Anup", _lastName: "Mulay")
}
```



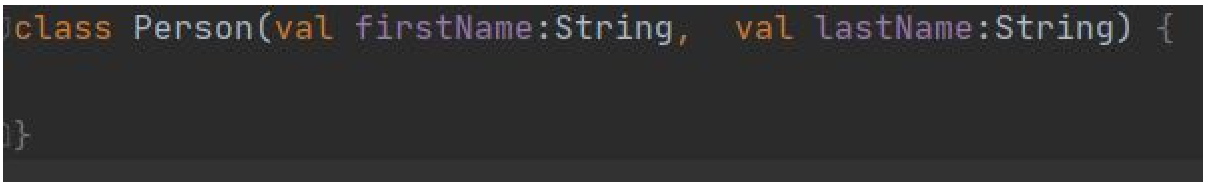
```
class Person(_firstName:String, _lastName:String) {

    val firstName:String
    val lastName:String

    init {
        firstName = _firstName
        lastName = _lastName
    }
}
```


```
class Person(_firstName:String, _lastName:String) {

    val firstName:String = _firstName
    val lastName:String = _lastName


}
```

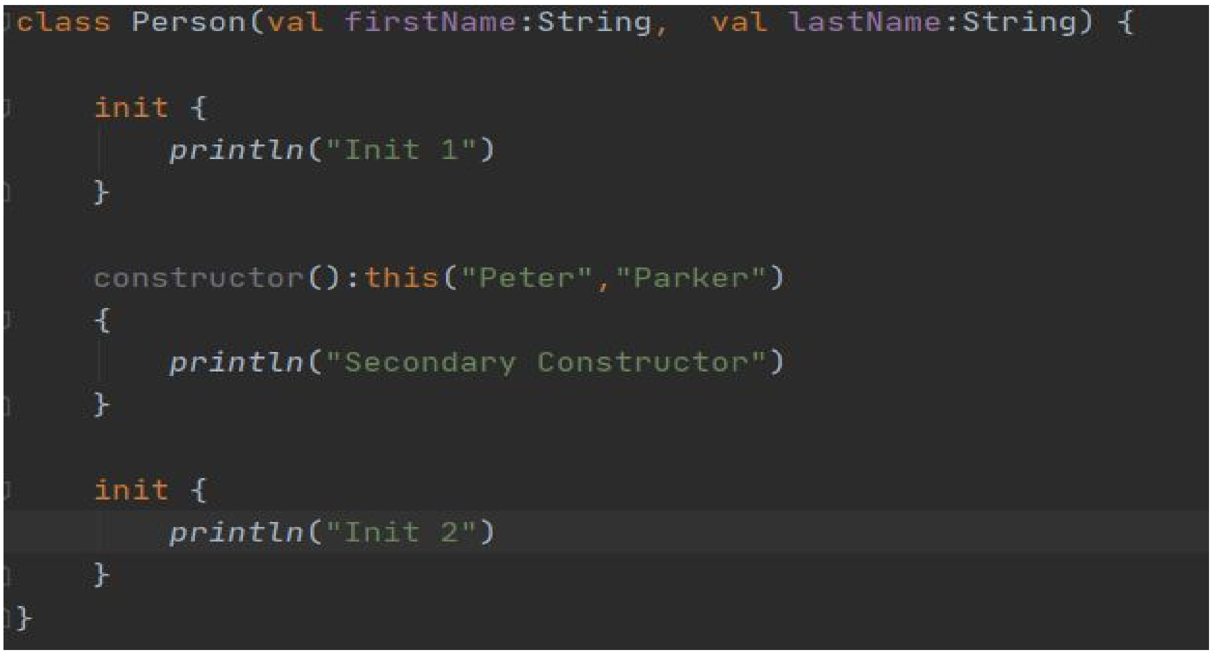
22. Property Access Syntax / No Getters and Setters

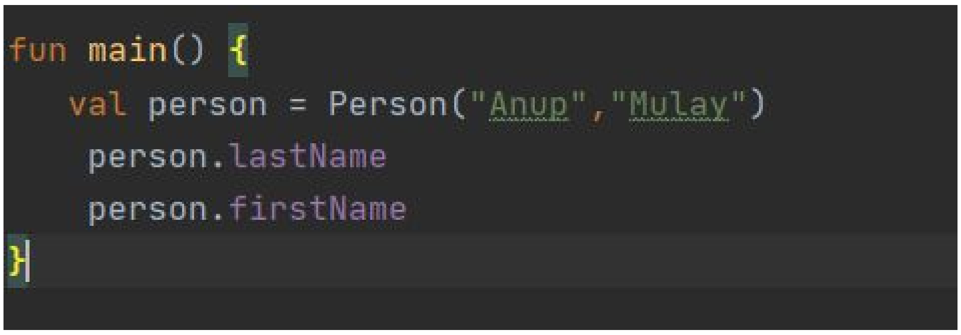

```
fun main() {
    val person = Person( _firstName: "Anup", _lastName: "Mulay")
     person.lastName
     person.firstName
}
```

23. Can Directly declare properties in the constructor


```
class Person(val firstName:String,  val lastName:String) {

}
```

24. Secondary Constructor

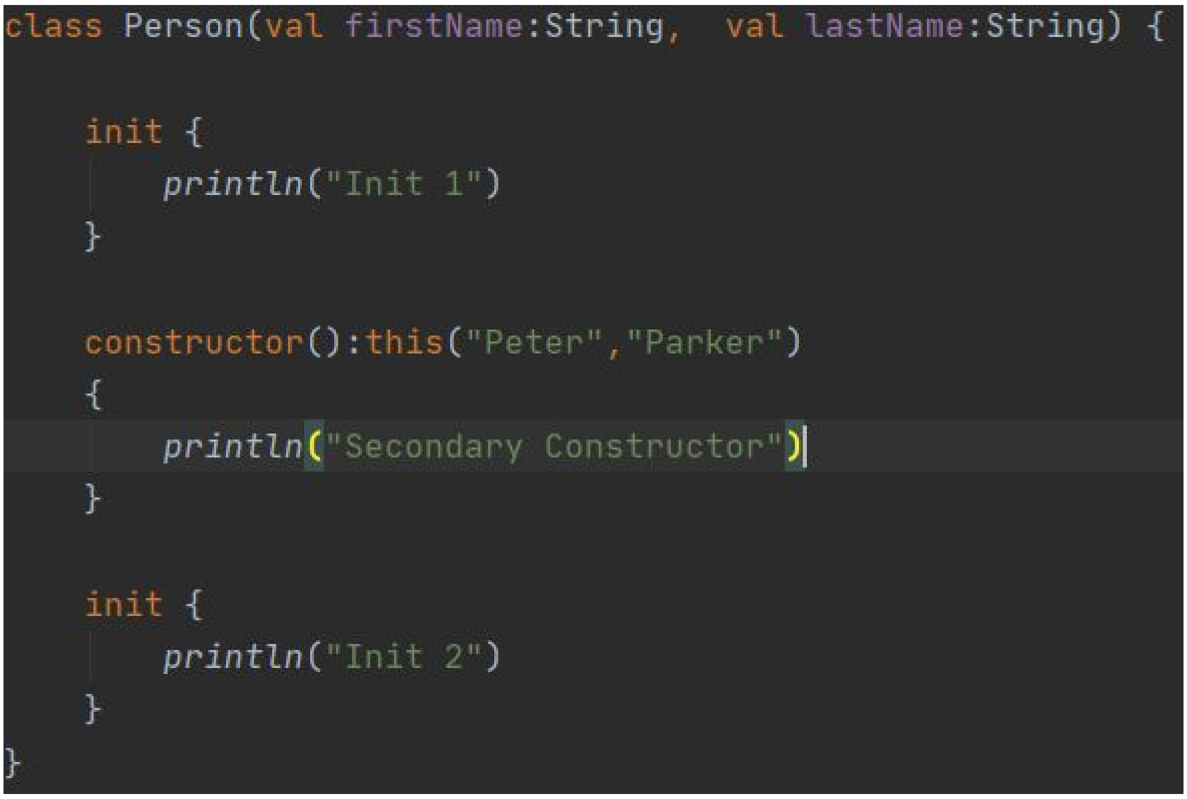

```
class Person(val firstName:String,  val lastName:String) {

    init {
        println("Init 1")
    }

    constructor():this("Peter","Parker")
    {
        println("Secondary Constructor")
    }

    init {
        println("Init 2")
    }
}
```



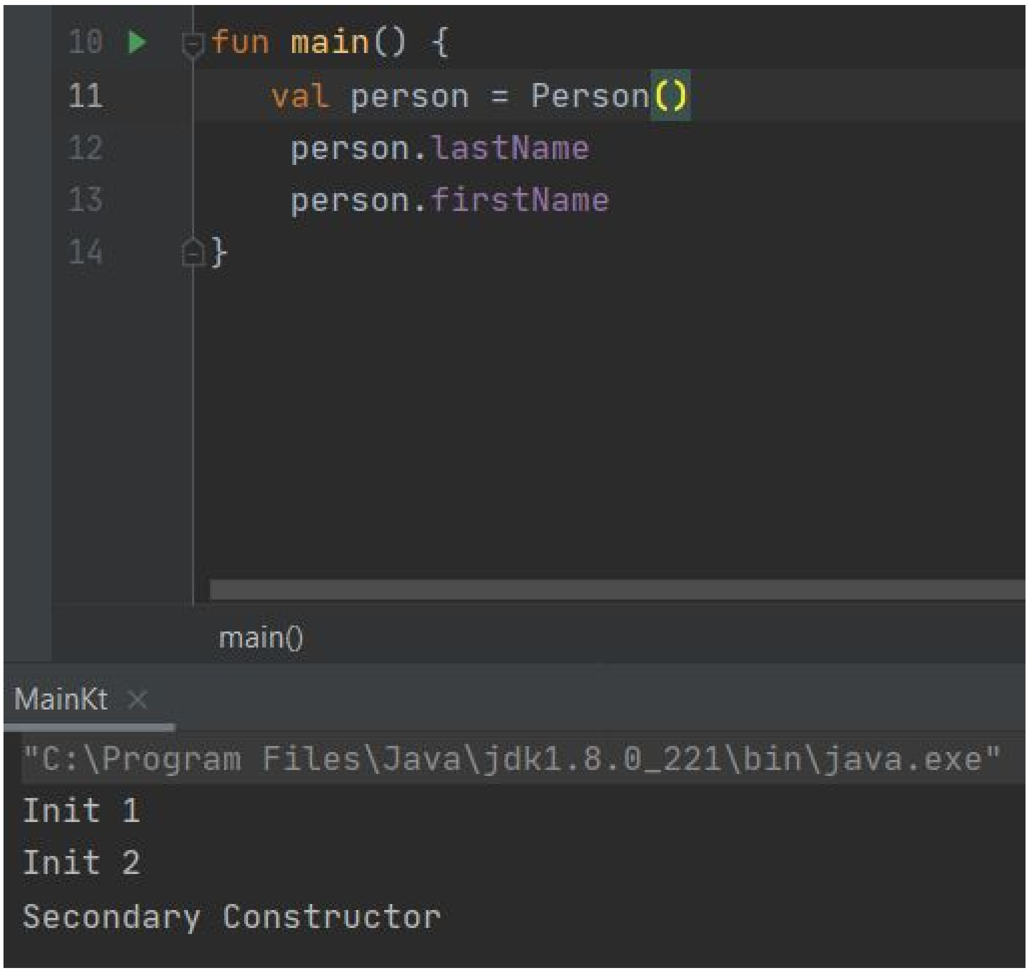
```kotlin
fun main() {
   val person = Person("Anup","Mulay")
    person.lastName
    person.firstName
}
```

In this case the secondary constructor will never get called and we will get output as :
Init 1
Init 2

25. Init blocks are always going to run before the secondary constructor



```kotlin
class Person(val firstName:String,  val lastName:String) {

    init {
        println("Init 1")
    }

    constructor():this("Peter","Parker")
    {
        println("Secondary Constructor")
    }

    init {
        println("Init 2")
    }
}
```



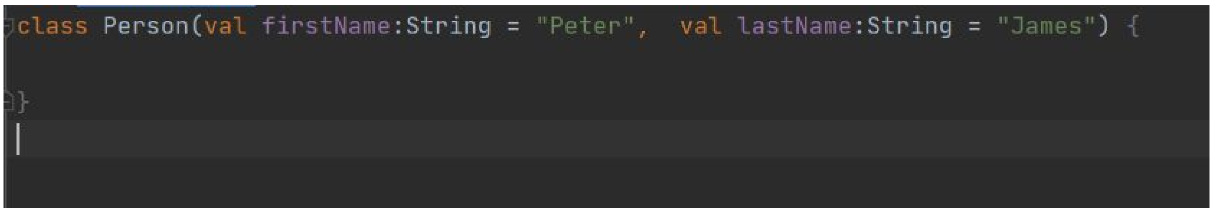
```
fun main() {
    val person = Person()
     person.lastName
     person.firstName
}
```

main()

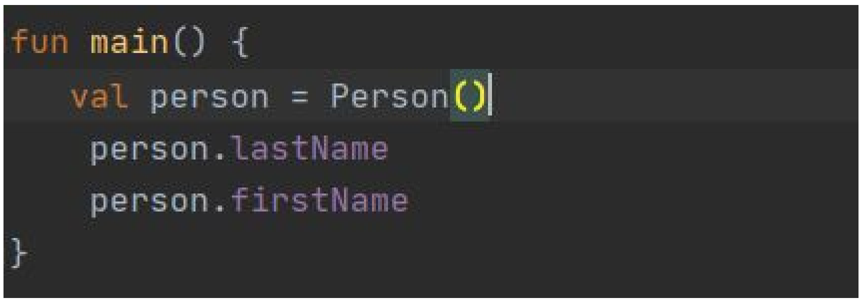
MainKt

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe"
Init 1
Init 2
Secondary Constructor
```

26. Secondary Constructor is not that used in real time application as we can give default values to parameters.



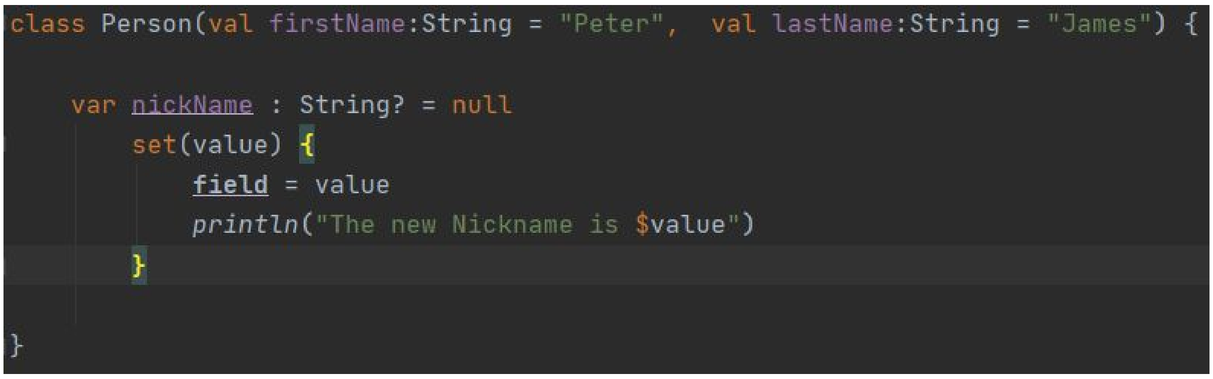
```
class Person(val firstName:String = "Peter",  val lastName:String = "James") {

}
```



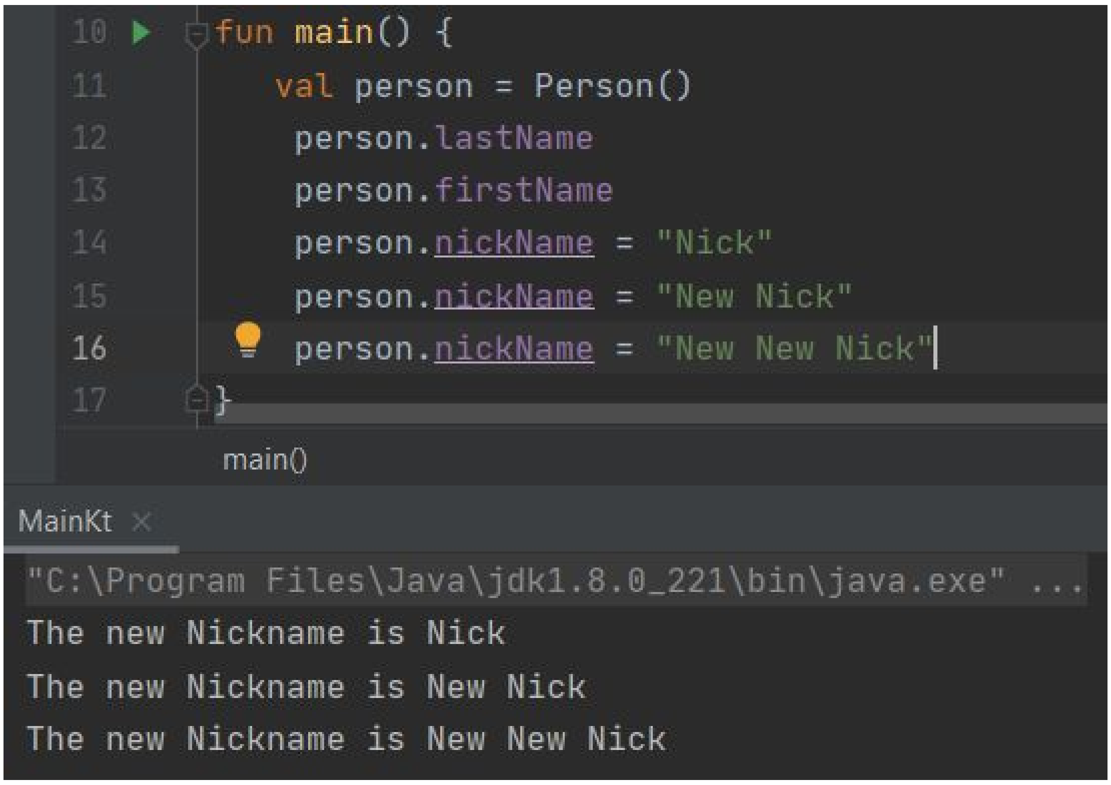
```
fun main() {
    val person = Person()
     person.lastName
     person.firstName
}
```

27. Closer look at class properties : In class person there are 2 properties defined in the primary constructor both of these are Read Only Properties so they don't have No Setters but only getters. Because of which we can leverage property access syntax in the Main file.

28. Properties in Kotlin will get getters and setters generated for them automatically by the compiler.
So, for val : getter Generated   and   var : Getter and Setter Generated
We can override the default behaviors of getters and setters to provide own implementations.
Nipping declaration
This set allows us to define the implementation when the set is called.
When we do this it will generate a backing field for this property.
So to assign the new value to nickname property we need to use a special keyword called field = value (If we dont do this then the actual value of nickname will never be updated)



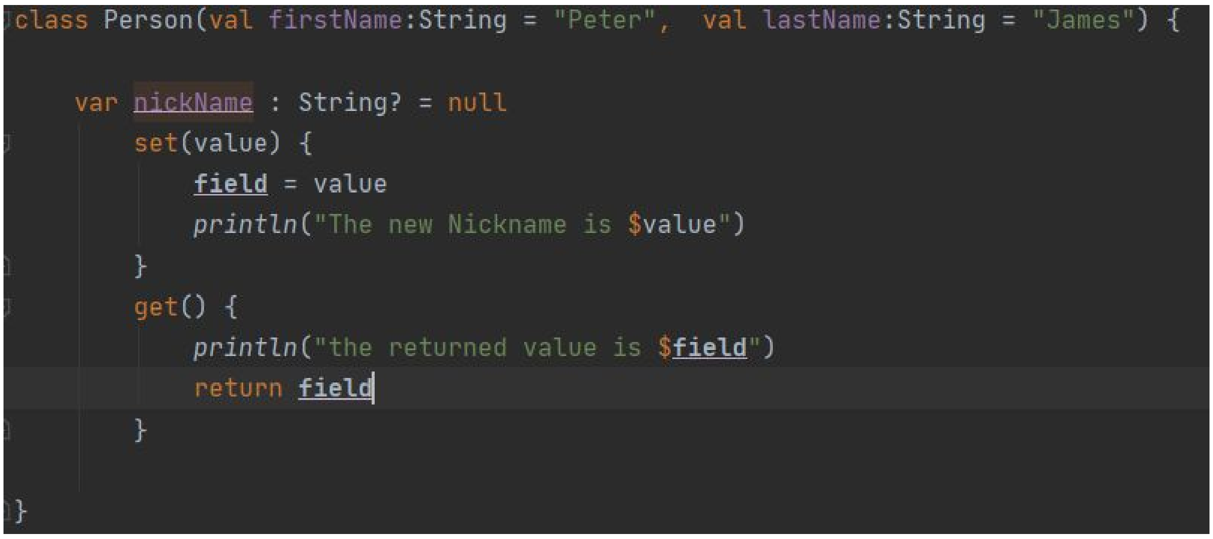
```kotlin
class Person(val firstName:String = "Peter",  val lastName:String = "James") {

    var nickName : String? = null
        set(value) {
            field = value
            println("The new Nickname is $value")
        }

}
```



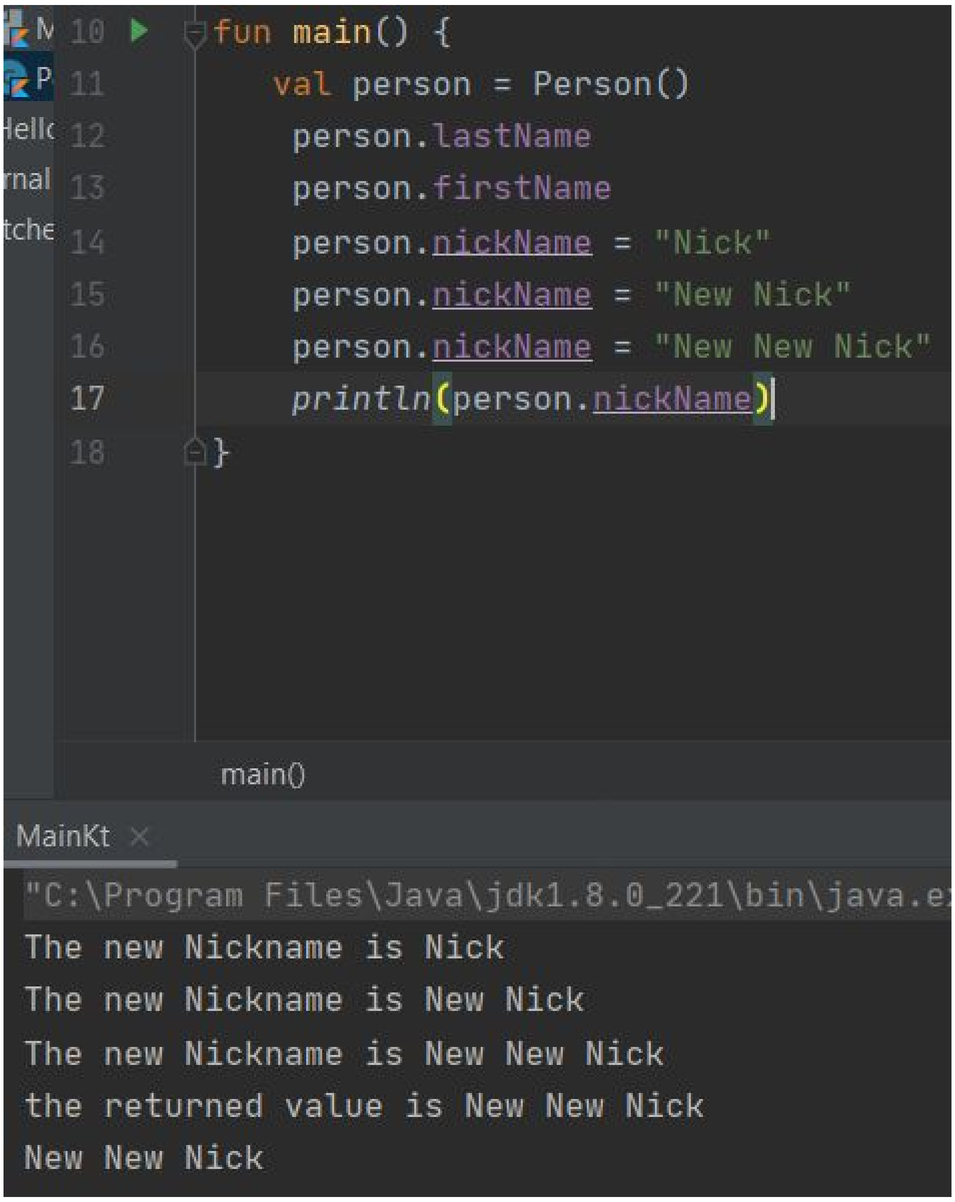
```kotlin
fun main() {
    val person = Person()
     person.lastName
     person.firstName
     person.nickName = "Nick"
     person.nickName = "New Nick"
     person.nickName = "New New Nick"
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
The new Nickname is Nick
The new Nickname is New Nick
The new Nickname is New New Nick
```

29. Same Applied to Getters



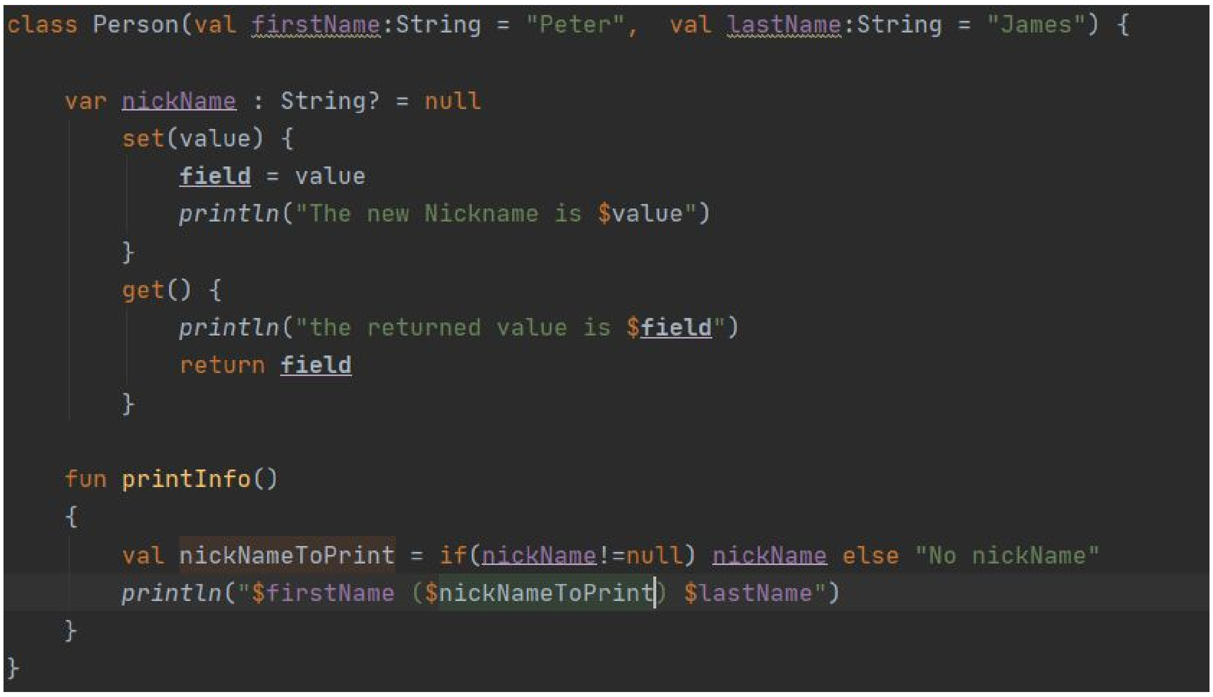
```
class Person(val firstName:String = "Peter",  val lastName:String = "James") {

    var nickName : String? = null
        set(value) {
            field = value
            println("The new Nickname is $value")
        }
        get() {
            println("the returned value is $field")
            return field
        }

}
```



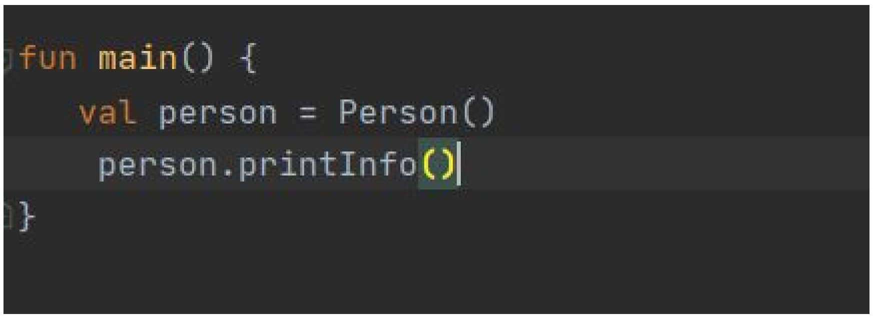
```
fun main() {
    val person = Person()
     person.lastName
     person.firstName
     person.nickName = "Nick"
     person.nickName = "New Nick"
     person.nickName = "New New Nick"
     println(person.nickName)
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.e
The new Nickname is Nick
The new Nickname is New Nick
The new Nickname is New New Nick
the returned value is New New Nick
New New Nick
```

30. Method in Class

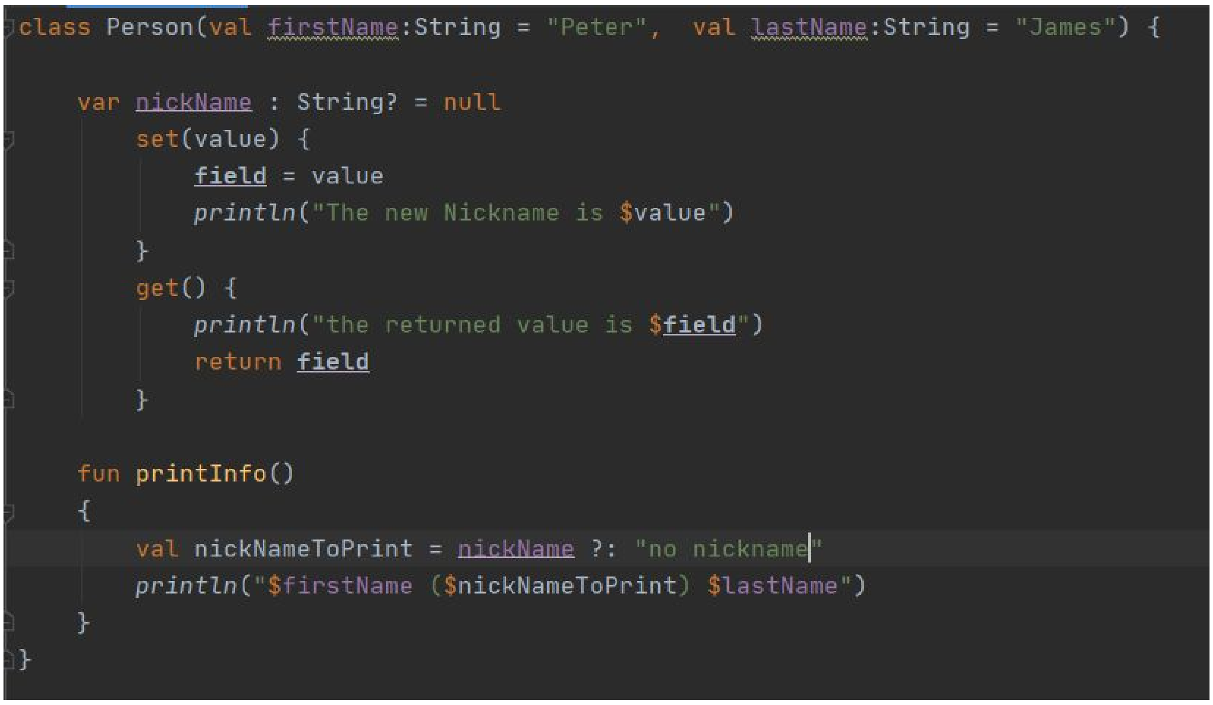


```kotlin
class Person(val firstName:String = "Peter",  val lastName:String = "James") {

    var nickName : String? = null
        set(value) {
            field = value
            println("The new Nickname is $value")
        }
        get() {
            println("the returned value is $field")
            return field
        }

    fun printInfo()
    {
        val nickNameToPrint = if(nickName!=null) nickName else "No nickName"
        println("$firstName ($nickNameToPrint) $lastName")
    }
}
```



```kotlin
fun main() {
    val person = Person()
     person.printInfo()
}
```

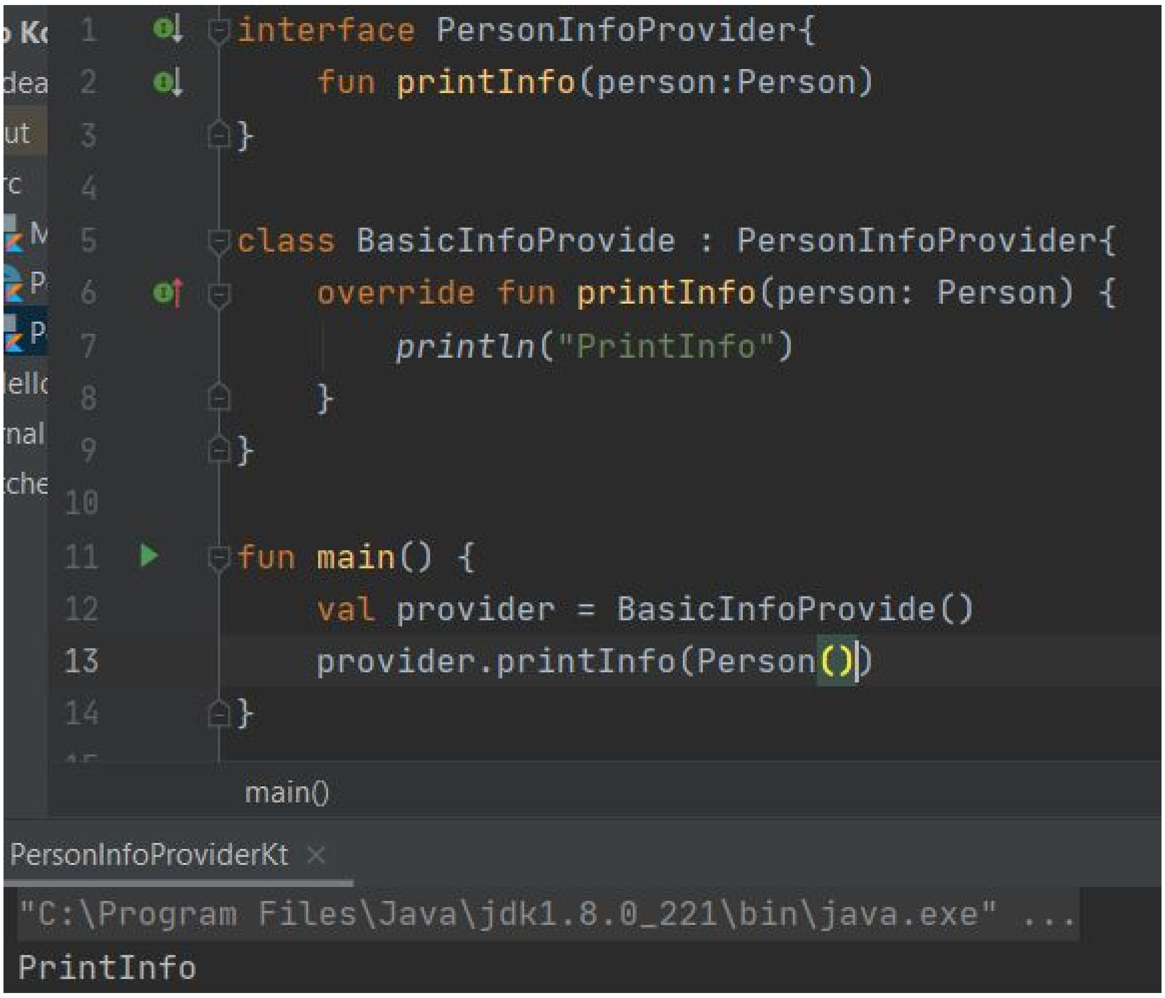
31. Simplification of above if condition : (?: known as Elvis Operator)



```kotlin
class Person(val firstName:String = "Peter",  val lastName:String = "James") {

    var nickName : String? = null
        set(value) {
            field = value
            println("The new Nickname is $value")
        }
        get() {
            println("the returned value is $field")
            return field
        }

    fun printInfo()
    {
        val nickNameToPrint = nickName ?: "no nickname"
        println("$firstName ($nickNameToPrint) $lastName")
    }
}
```

32. Visibility Operator in Kotlin
In Kotlin classes, properties, methods have Public visibility by default
Internal : the class is public which this module
Private : private is available only in the file in which it is implemented
Protected : protected property will be available in the same class it is defined are any of its subclasses

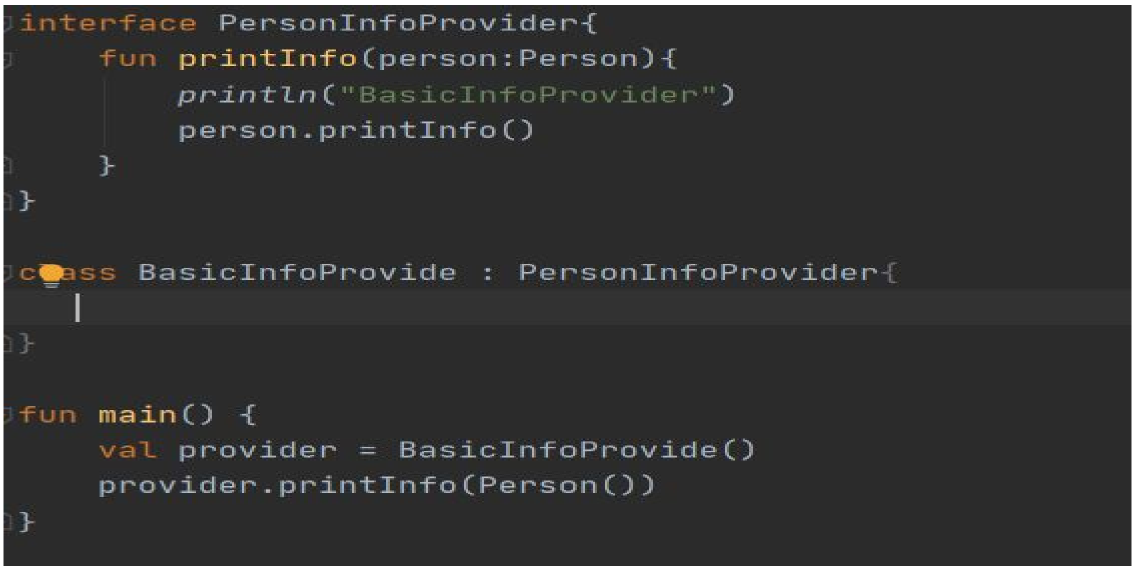
33. Interface in Kotlin
Support Marker Interface



```kotlin
interface PersonInfoProvider{
    fun printInfo(person:Person)
}

class BasicInfoProvide : PersonInfoProvider{
    override fun printInfo(person: Person) {
        println("PrintInfo")
    }
}

fun main() {
    val provider = BasicInfoProvide()
    provider.printInfo(Person())
}
```

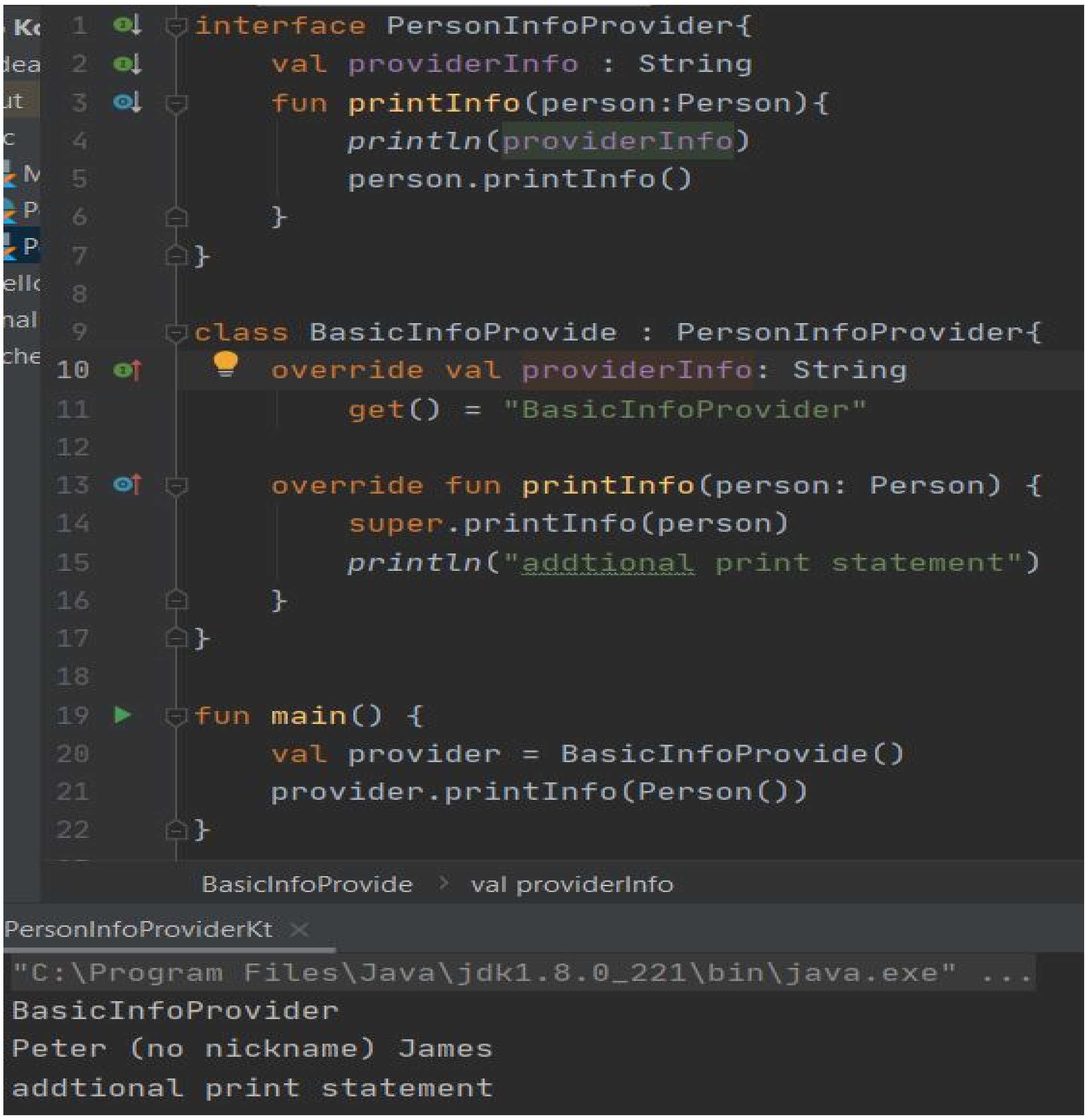
```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
PrintInfo
```

34. Interface with default implementation



```
interface PersonInfoProvider{
    fun printInfo(person:Person){
        println("BasicInfoProvider")
        person.printInfo()
    }
}

class BasicInfoProvide : PersonInfoProvider{

}

fun main() {
    val provider = BasicInfoProvide()
    provider.printInfo(Person())
}
```

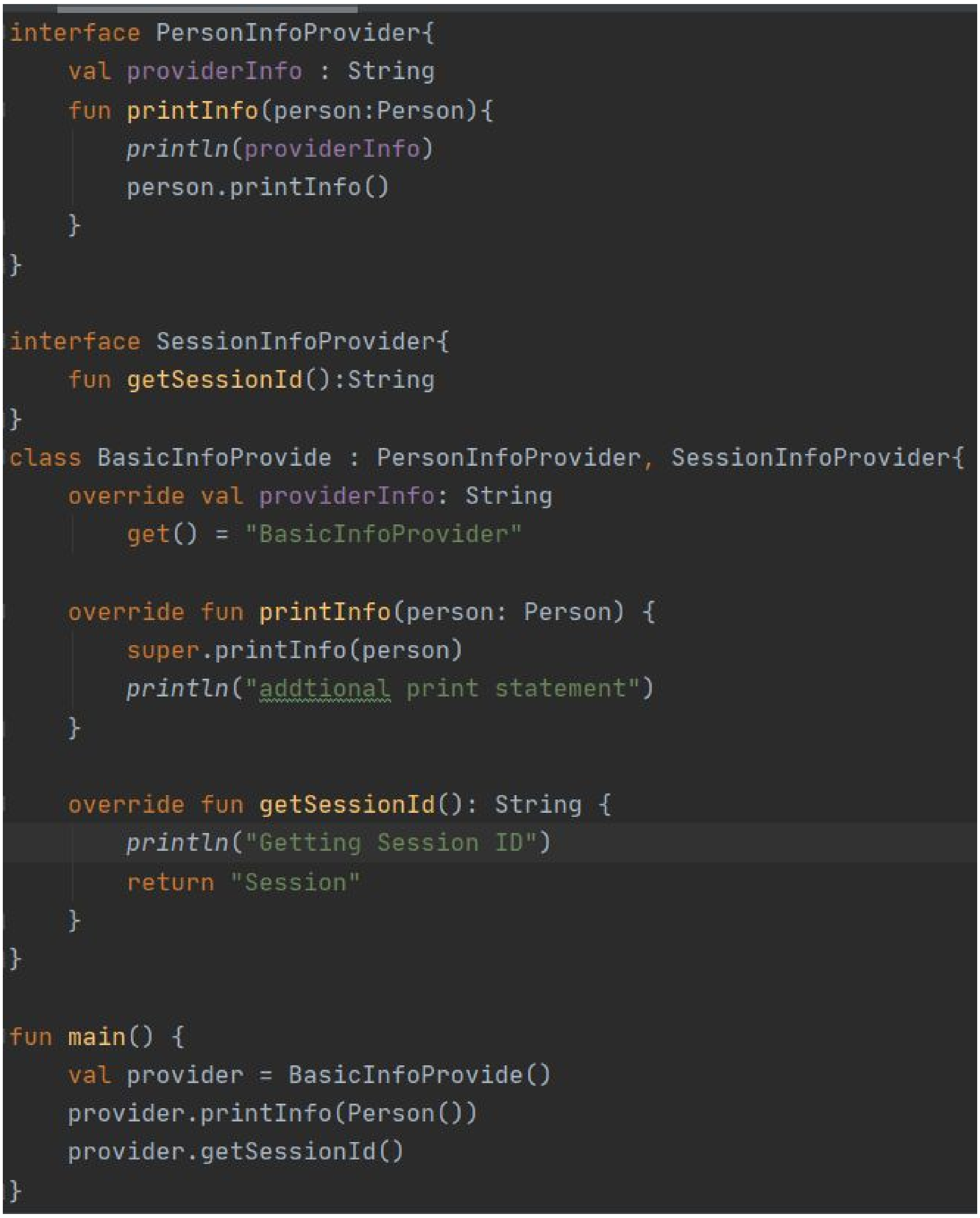
35. Property initializers are not allowed in the interface Property has to be overriden



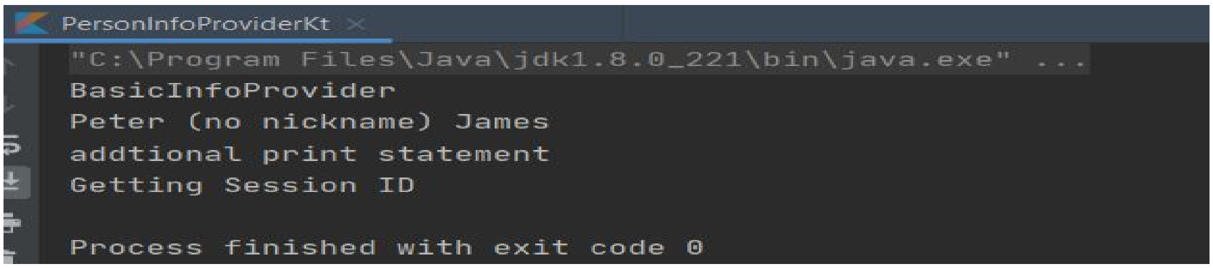
```
interface PersonInfoProvider{
    val providerInfo : String
    fun printInfo(person:Person){
        println(providerInfo)
        person.printInfo()
    }
}

class BasicInfoProvide : PersonInfoProvider{
    override val providerInfo: String
        get() = "BasicInfoProvider"

    override fun printInfo(person: Person) {
        super.printInfo(person)
        println("addtional print statement")
    }
}

fun main() {
    val provider = BasicInfoProvide()
    provider.printInfo(Person())
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
BasicInfoProvider
Peter (no nickname) James
addtional print statement
```

36. Implementation of multiple interfaces by a class



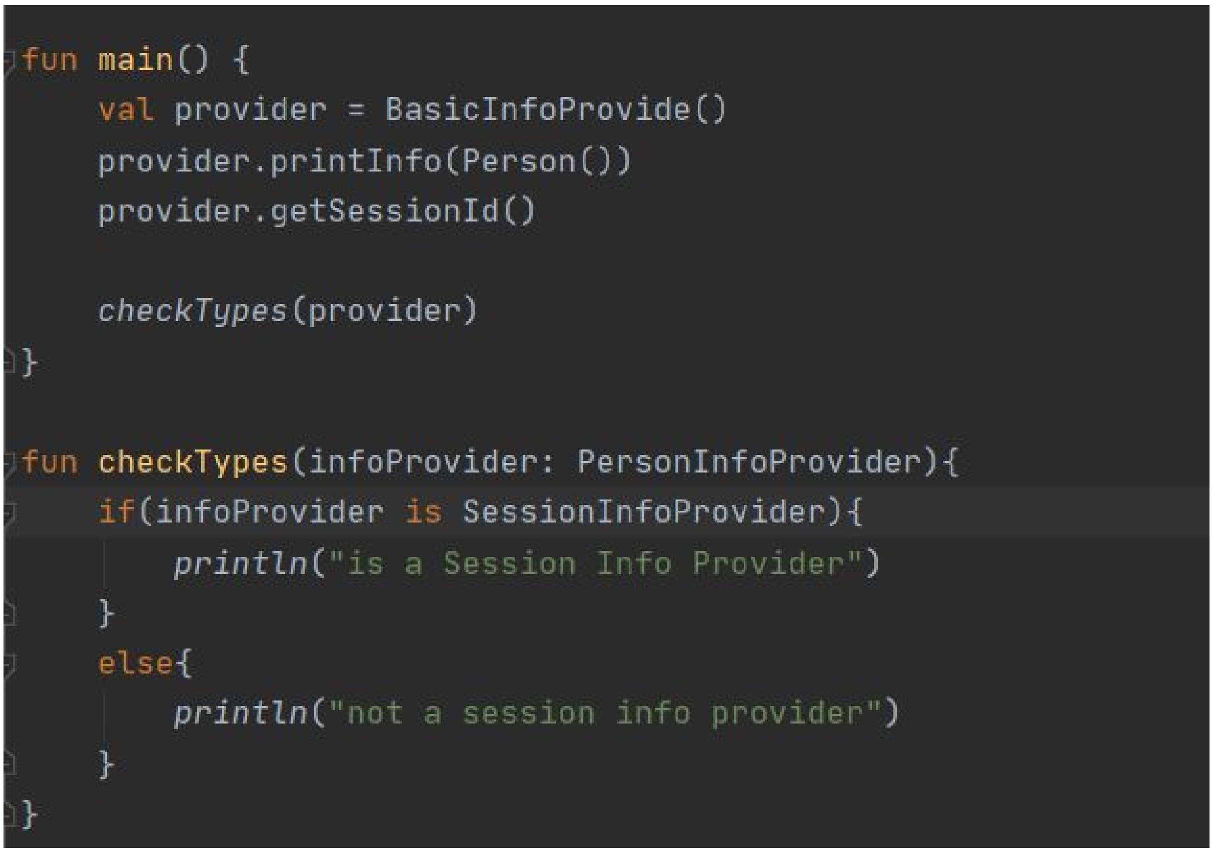
```kotlin
interface PersonInfoProvider{
    val providerInfo : String
    fun printInfo(person:Person){
        println(providerInfo)
        person.printInfo()
    }
}

interface SessionInfoProvider{
    fun getSessionId():String
}
class BasicInfoProvide : PersonInfoProvider, SessionInfoProvider{
    override val providerInfo: String
        get() = "BasicInfoProvider"

    override fun printInfo(person: Person) {
        super.printInfo(person)
        println("addtional print statement")
    }

    override fun getSessionId(): String {
        println("Getting Session ID")
        return "Session"
    }
}

fun main() {
    val provider = BasicInfoProvide()
    provider.printInfo(Person())
    provider.getSessionId()
}
```



PersonInfoProviderKt

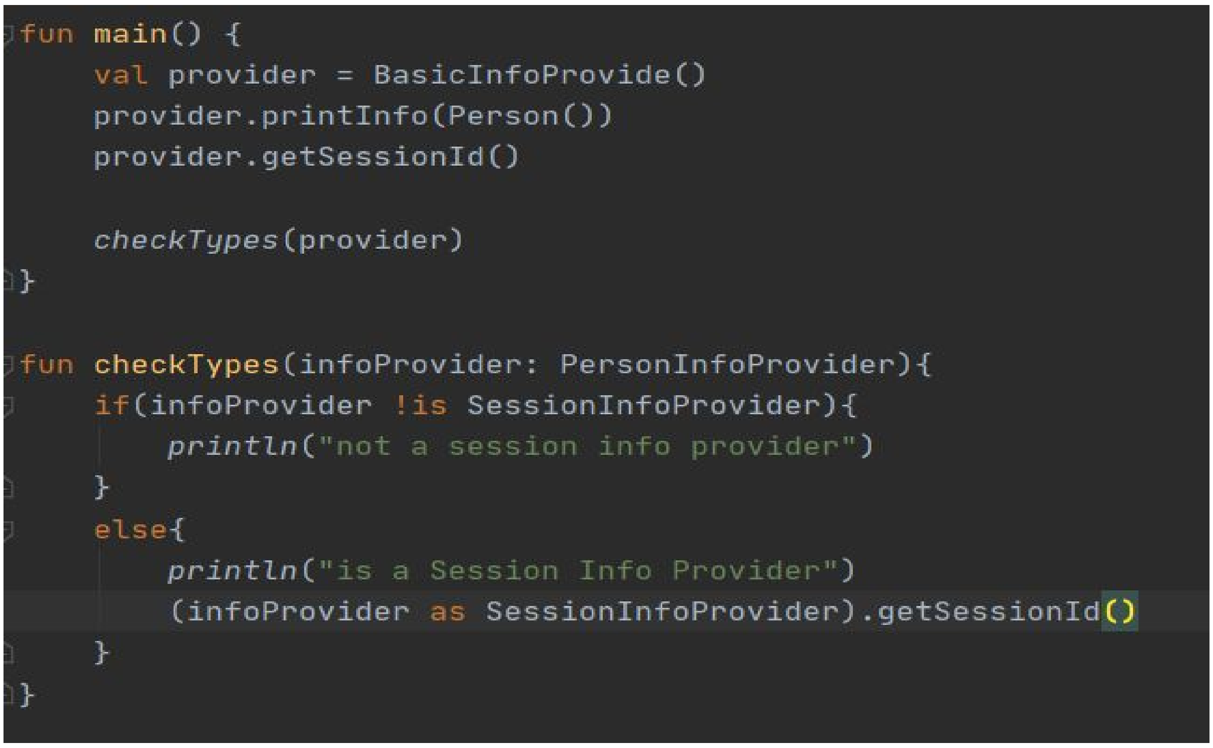
```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
BasicInfoProvider
Peter (no nickname) James
addtional print statement
Getting Session ID

Process finished with exit code 0
```

37. Type Checking and Type Casting in kotlin
Type Checking :



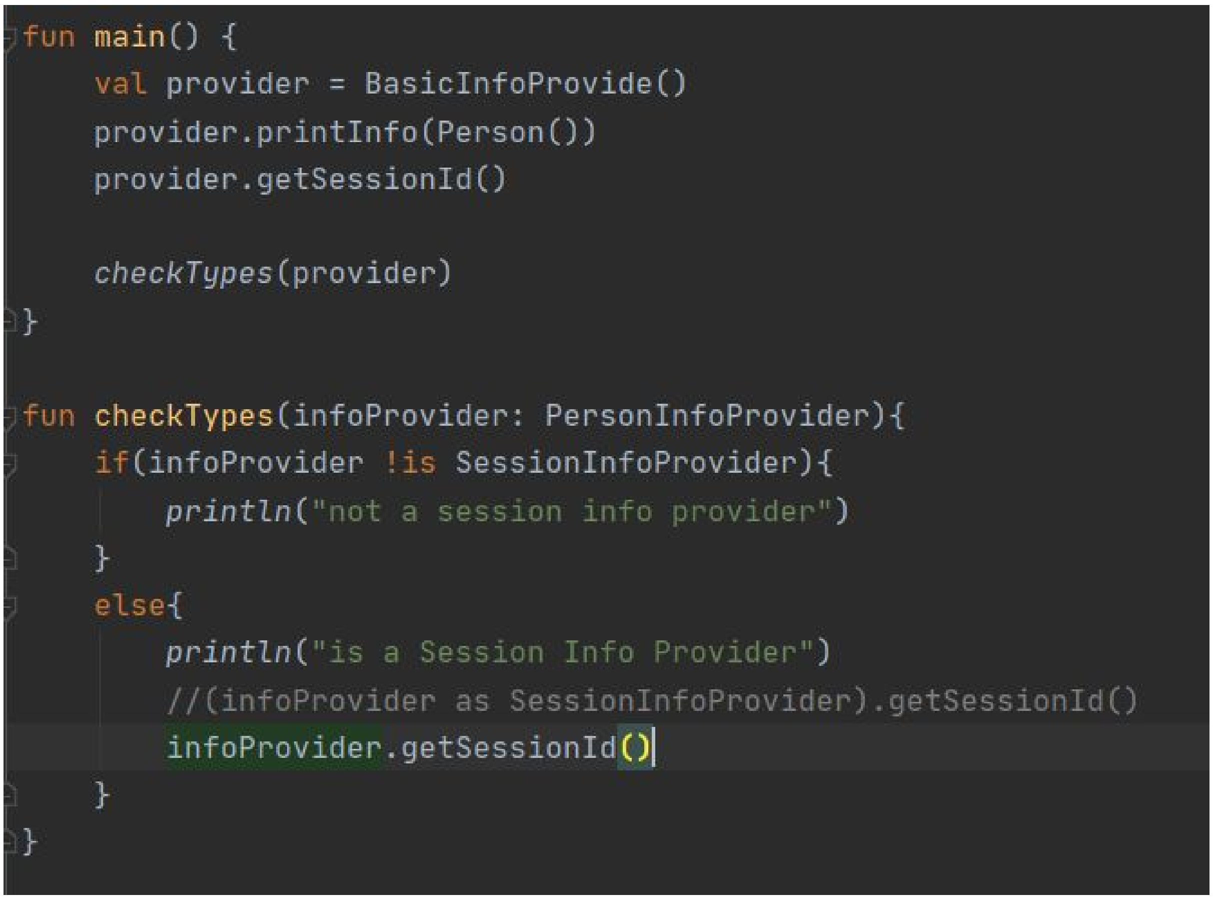
```kotlin
fun main() {
    val provider = BasicInfoProvide()
    provider.printInfo(Person())
    provider.getSessionId()

    checkTypes(provider)
}

fun checkTypes(infoProvider: PersonInfoProvider){
    if(infoProvider is SessionInfoProvider){
        println("is a Session Info Provider")
    }
    else{
        println("not a session info provider")
    }
}
```

Type Casting :



```kotlin
fun main() {
    val provider = BasicInfoProvide()
    provider.printInfo(Person())
    provider.getSessionId()

    checkTypes(provider)
}

fun checkTypes(infoProvider: PersonInfoProvider){
    if(infoProvider !is SessionInfoProvider){
        println("not a session info provider")
    }
    else{
        println("is a Session Info Provider")
        (infoProvider as SessionInfoProvider).getSessionId()
    }
}
```

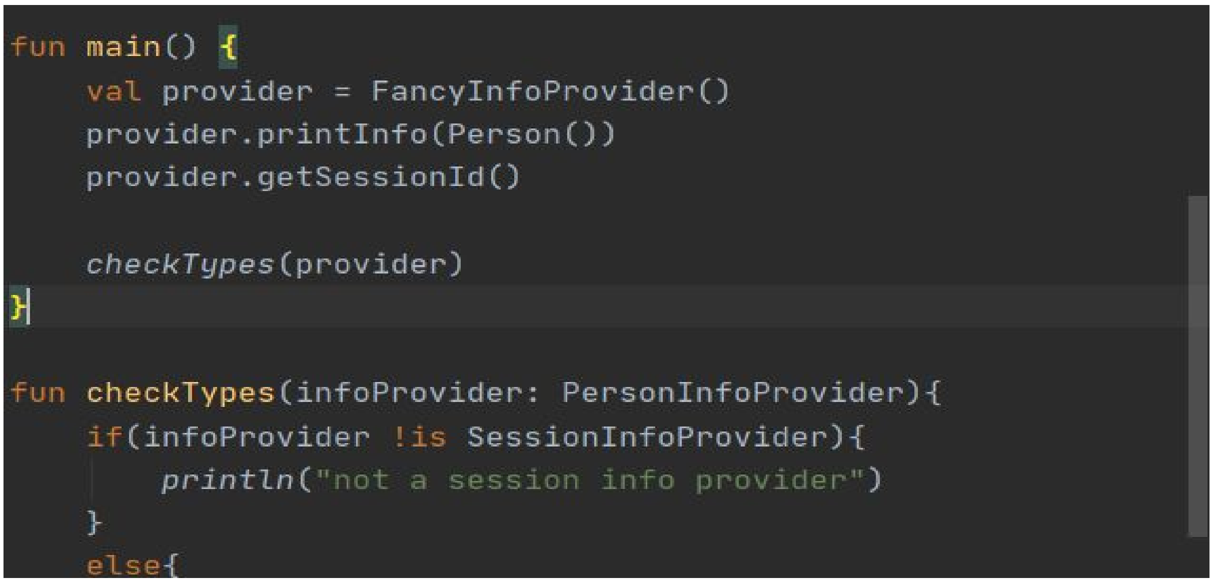
Kotlin also provides the Smart Casting :
If the compiler can check the type and validate that the type will not change. Then you don't need to do any additional casting.



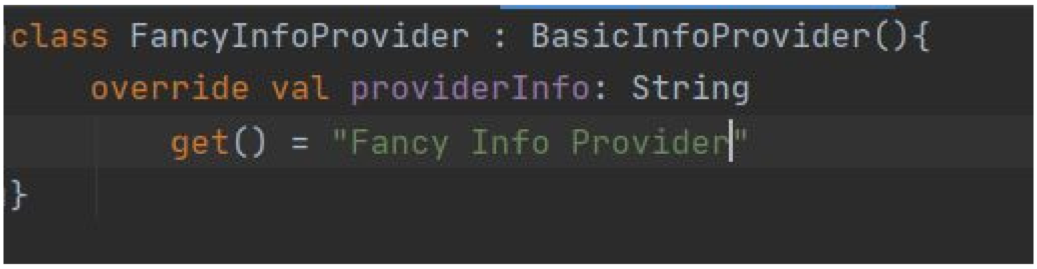
```
fun main() {
    val provider = BasicInfoProvide()
    provider.printInfo(Person())
    provider.getSessionId()

    checkTypes(provider)
}

fun checkTypes(infoProvider: PersonInfoProvider){
    if(infoProvider !is SessionInfoProvider){
        println("not a session info provider")
    }
    else{
        println("is a Session Info Provider")
        //(infoProvider as SessionInfoProvider).getSessionId()
        infoProvider.getSessionId()
    }
}
```

Here the compiler performs the smart casting

38. Inheritance in Kotlin
 Classes in Kotlin have specific characteristics : By default the classes are closed meaning the class cannot be extended. To make the class extendable we need to provide the open keyword to the class declaration.

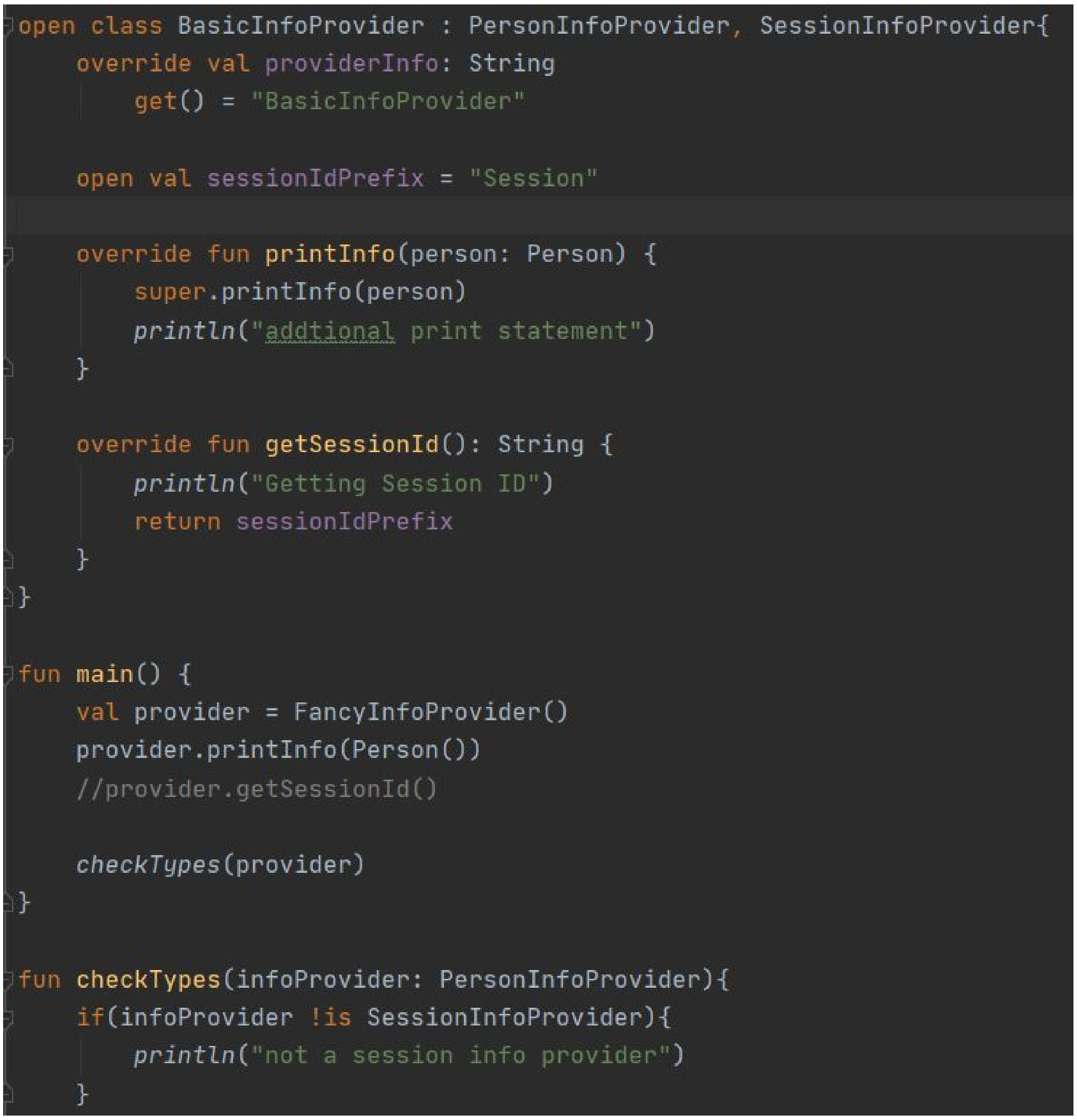


```
fun main() {
    val provider = FancyInfoProvider()
    provider.printInfo(Person())
    provider.getSessionId()

    checkTypes(provider)
}

fun checkTypes(infoProvider: PersonInfoProvider){
    if(infoProvider !is SessionInfoProvider){
        println("not a session info provider")
    }
    else{
```

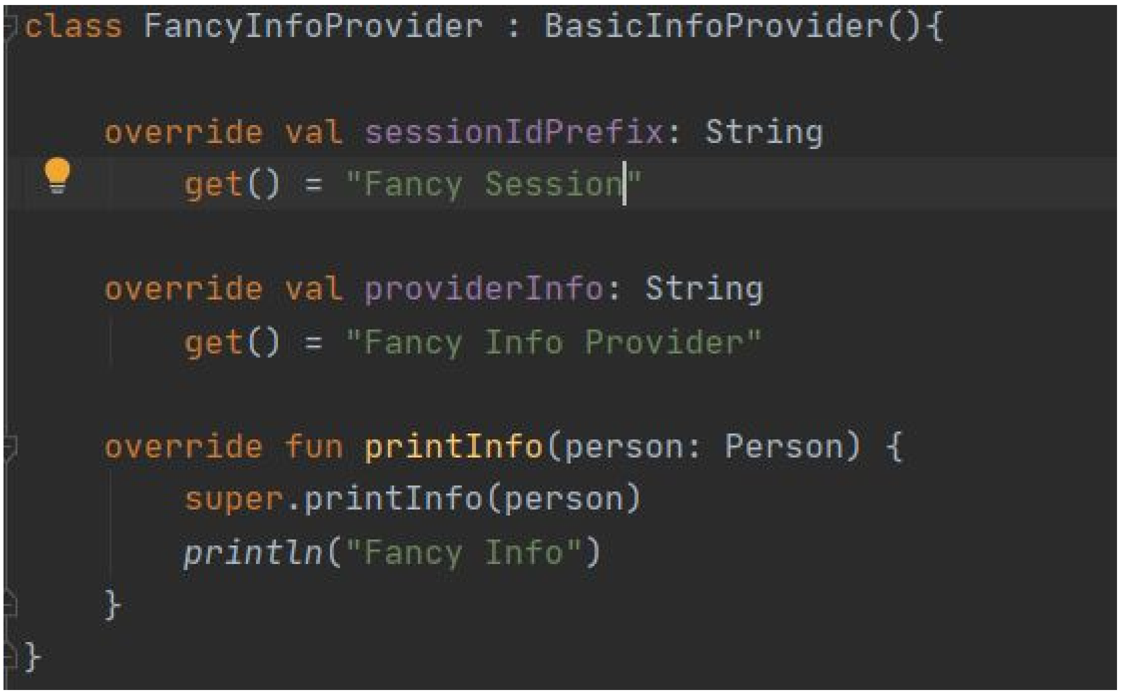


```kotlin
class FancyInfoProvider : BasicInfoProvider(){
    override val providerInfo: String
        get() = "Fancy Info Provider"
}
```

39. Kotlin forces you to mark both your classes, properties and methods being explicitly ‘open’ to extension.

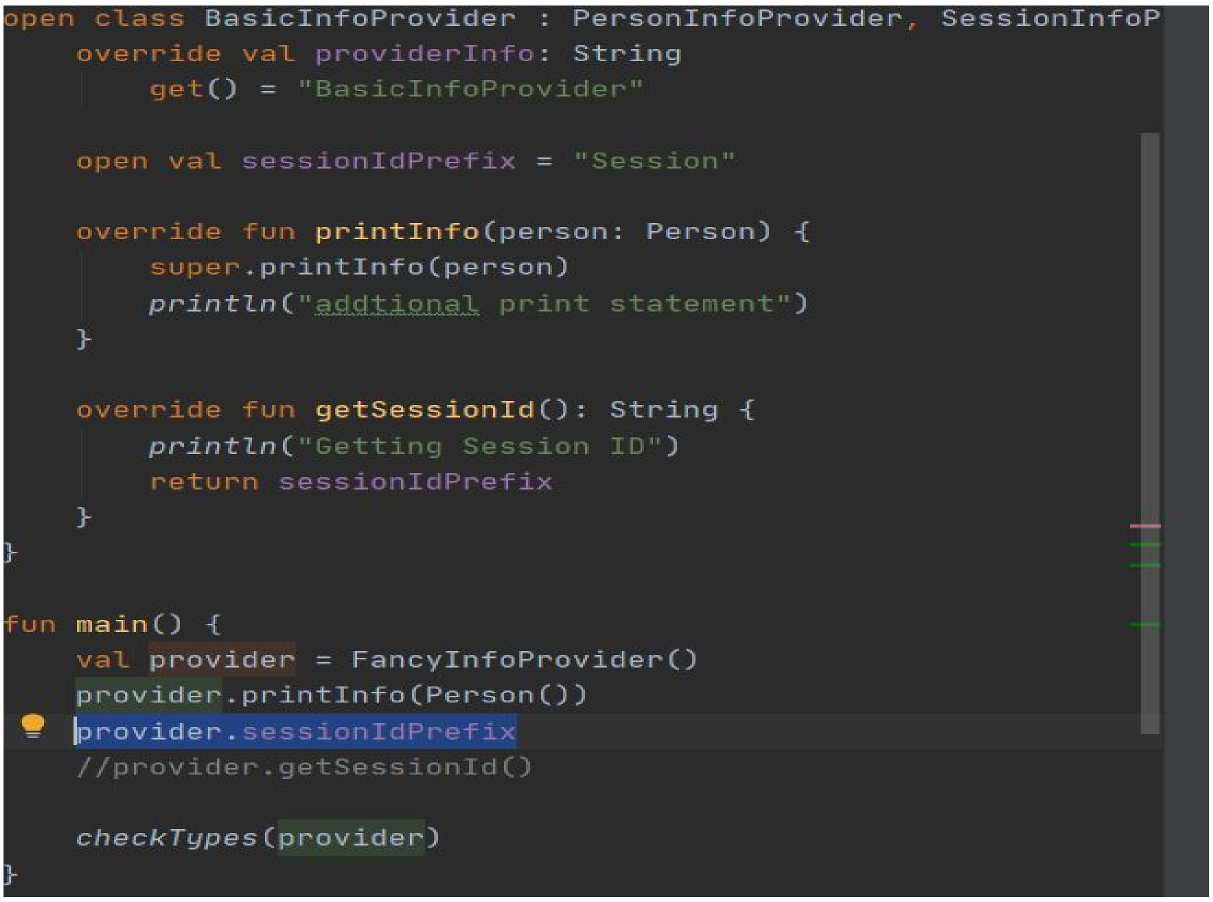


```kotlin
open class BasicInfoProvider : PersonInfoProvider, SessionInfoProvider{
    override val providerInfo: String
        get() = "BasicInfoProvider"

    open val sessionIdPrefix = "Session"


    override fun printInfo(person: Person) {
        super.printInfo(person)
        println("addtional print statement")
    }

    override fun getSessionId(): String {
        println("Getting Session ID")
        return sessionIdPrefix
    }
}

fun main() {
    val provider = FancyInfoProvider()
    provider.printInfo(Person())
    //provider.getSessionId()

    checkTypes(provider)
}

fun checkTypes(infoProvider: PersonInfoProvider){
    if(infoProvider !is SessionInfoProvider){
        println("not a session info provider")
    }
```

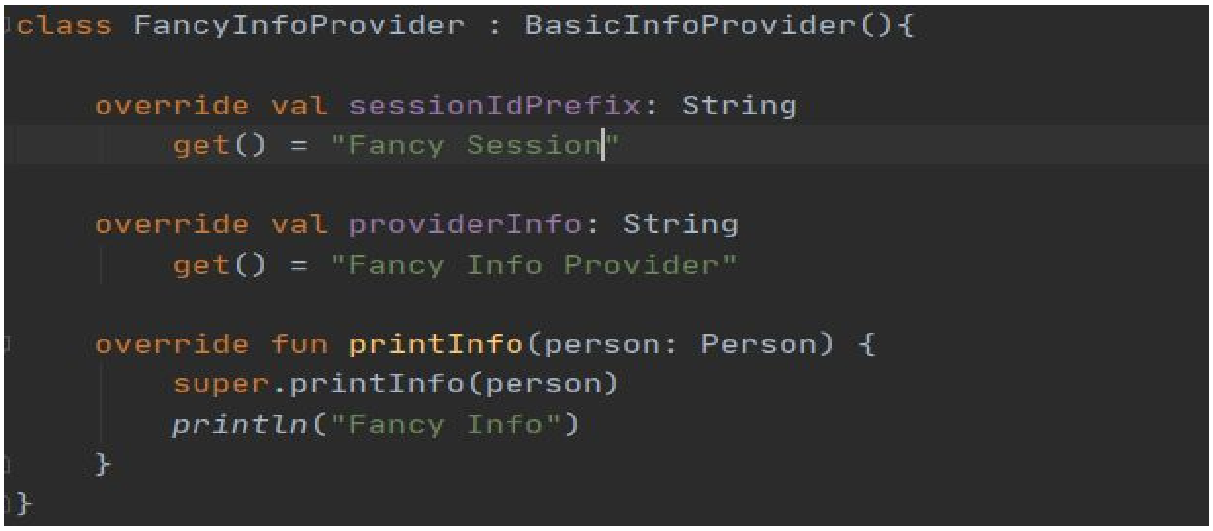


```kotlin
class FancyInfoProvider : BasicInfoProvider(){

    override val sessionIdPrefix: String
        get() = "Fancy Session"

    override val providerInfo: String
        get() = "Fancy Info Provider"

    override fun printInfo(person: Person) {
        super.printInfo(person)
        println("Fancy Info")
    }
}
```

In such case we encounter a problem,
Now we can have the ‘open’ marked sessionIdPrefix directly available in the main() which should not be the case (sessionIdPrefix should be the implementation detail of the class FancyInfoProvider). In short our API should not expose such a property, but it happens because the is marked open (Public) property.
To resolve this we can make the sessionIdPrefix variable ‘protected open’

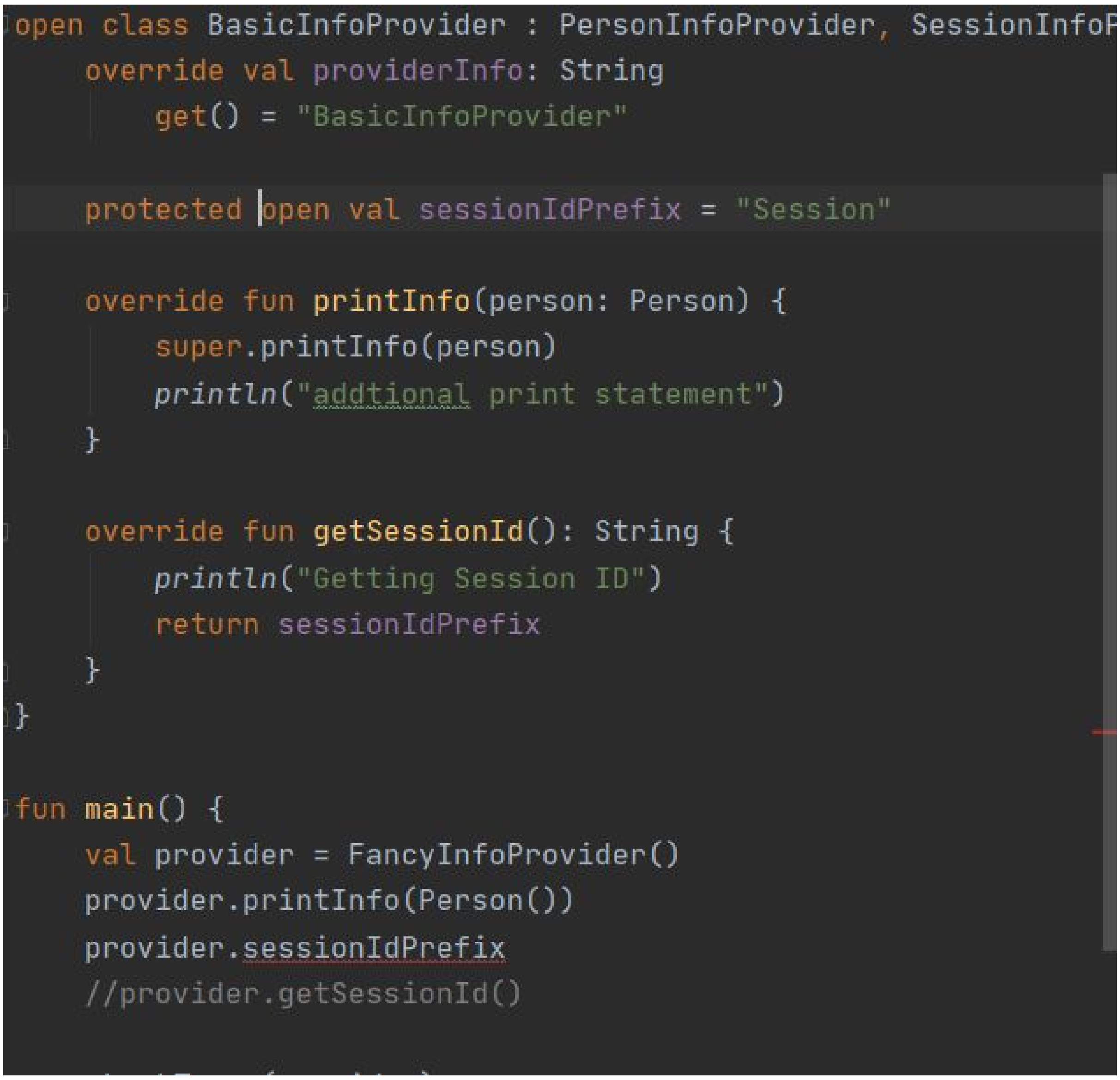


```kotlin
open class BasicInfoProvider : PersonInfoProvider, SessionInfoP
    override val providerInfo: String
        get() = "BasicInfoProvider"

    open val sessionIdPrefix = "Session"

    override fun printInfo(person: Person) {
        super.printInfo(person)
        println("addtional print statement")
    }

    override fun getSessionId(): String {
        println("Getting Session ID")
        return sessionIdPrefix
    }
}

fun main() {
    val provider = FancyInfoProvider()
    provider.printInfo(Person())
    provider.sessionIdPrefix
    //provider.getSessionId()

    checkTypes(provider)
}
```



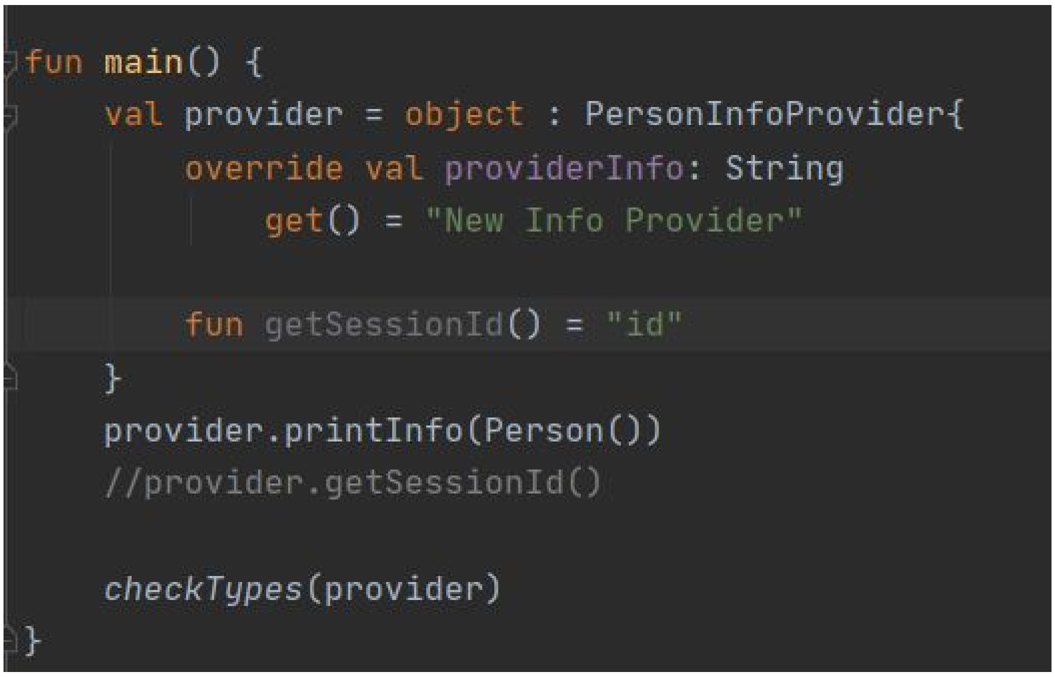
```
class FancyInfoProvider : BasicInfoProvider(){

    override val sessionIdPrefix: String
        get() = "Fancy Session"

    override val providerInfo: String
        get() = "Fancy Info Provider"

    override fun printInfo(person: Person) {
        super.printInfo(person)
        println("Fancy Info")
    }
}
```

Resolution:



```
open class BasicInfoProvider : PersonInfoProvider, SessionInfoP
    override val providerInfo: String
        get() = "BasicInfoProvider"

    protected open val sessionIdPrefix = "Session"

    override fun printInfo(person: Person) {
        super.printInfo(person)
        println("addtional print statement")
    }

    override fun getSessionId(): String {
        println("Getting Session ID")
        return sessionIdPrefix
    }
}

fun main() {
    val provider = FancyInfoProvider()
    provider.printInfo(Person())
    provider.sessionIdPrefix
    //provider.getSessionId()
```

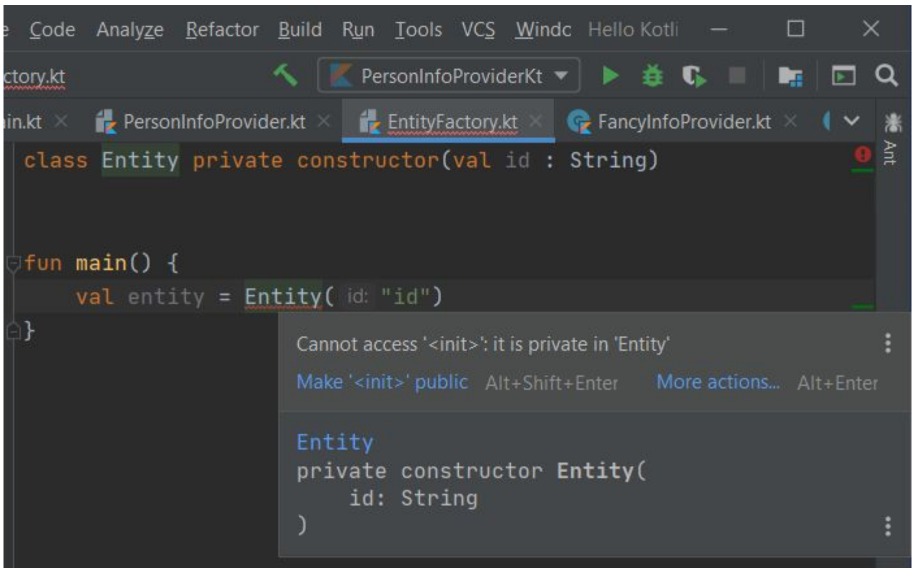
40. Instance Of Anonymous Inner Class using Object Expression



```
fun main() {
    val provider = object : PersonInfoProvider{
        override val providerInfo: String
            get() = "New Info Provider"

        fun getSessionId() = "id"
    }
    provider.printInfo(Person())
    //provider.getSessionId()

    checkTypes(provider)
}
```

This is Similar to JAVA it can be used for clickListener

41. Companion Object
An object scoped to an instance of another class.
Trying to create the factory to create instances of Entity class

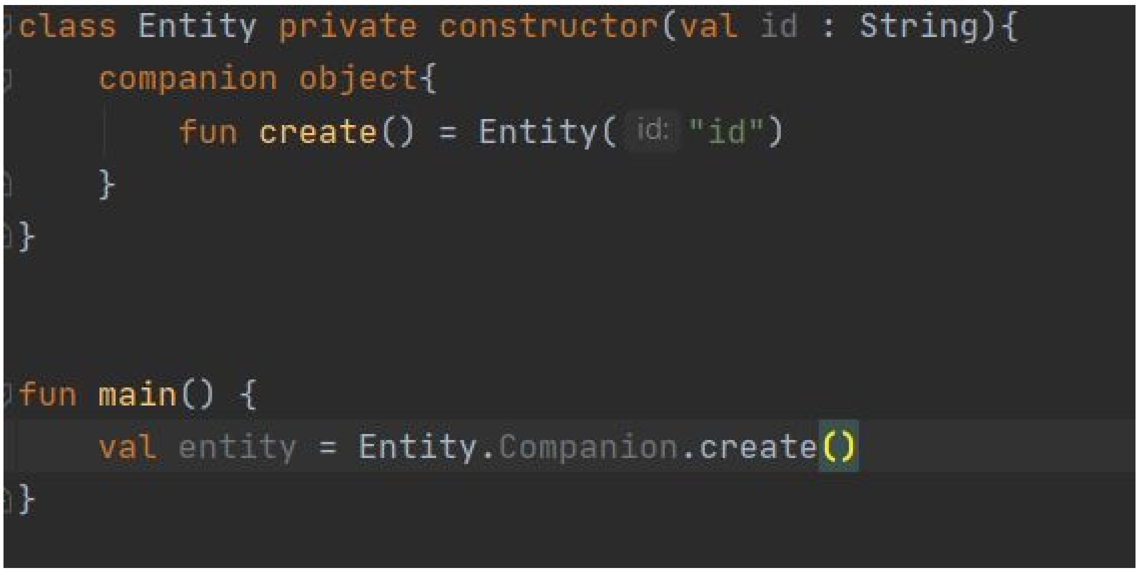


Code Analyze Refactor Build Run Tools VCS Windc Hello Kotli
ctory.kt
PersonInfoProviderKt
in.kt PersonInfoProvider.kt EntityFactory.kt FancyInfoProvider.kt
Ant

```
class Entity private constructor(val id : String)


fun main() {
    val entity = Entity( id: "id")
}
```

```
Cannot access '<init>': it is private in 'Entity'
Make '<init>' public  Alt+Shift+Enter    More actions...  Alt+Enter

Entity
private constructor Entity(
    id: String
)
```

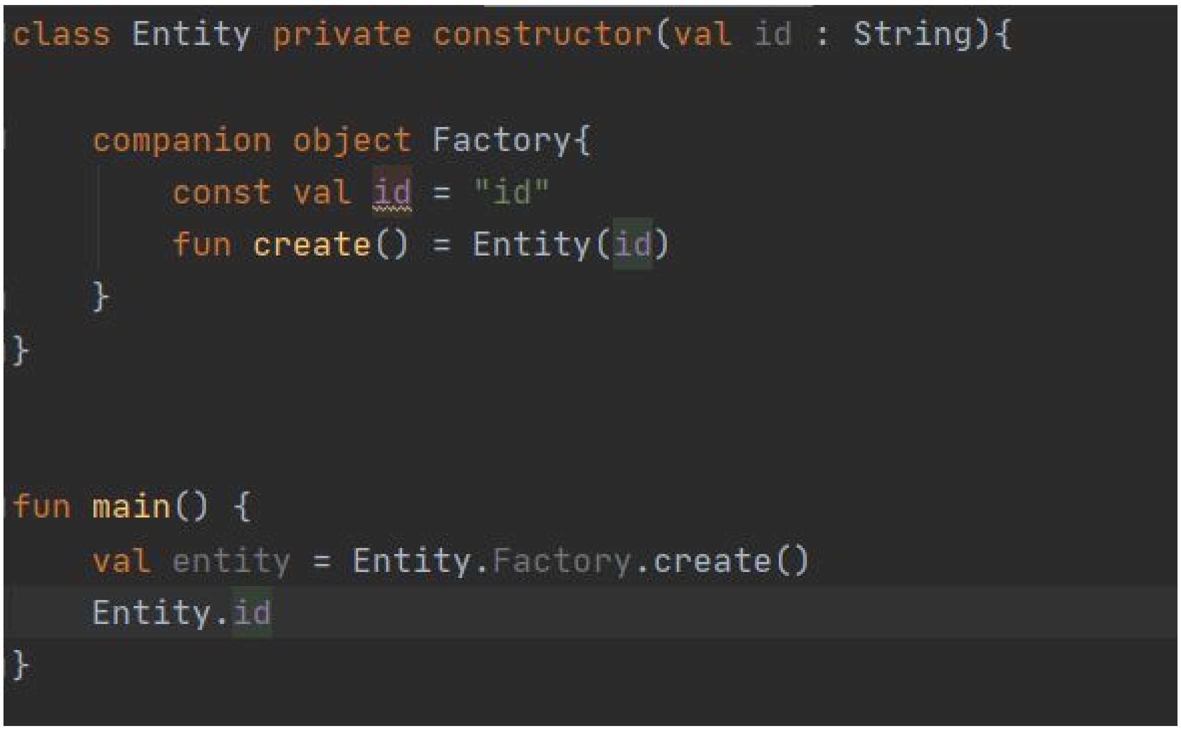
This happens due to the private constructor
This issue is resolved using the companion object; it an object scooped to an instance of another class.



```kotlin
class Entity private constructor(val id : String){
    companion object{
        fun create() = Entity( id: "id")
    }
}


fun main() {
    val entity = Entity.Companion.create()
}
```

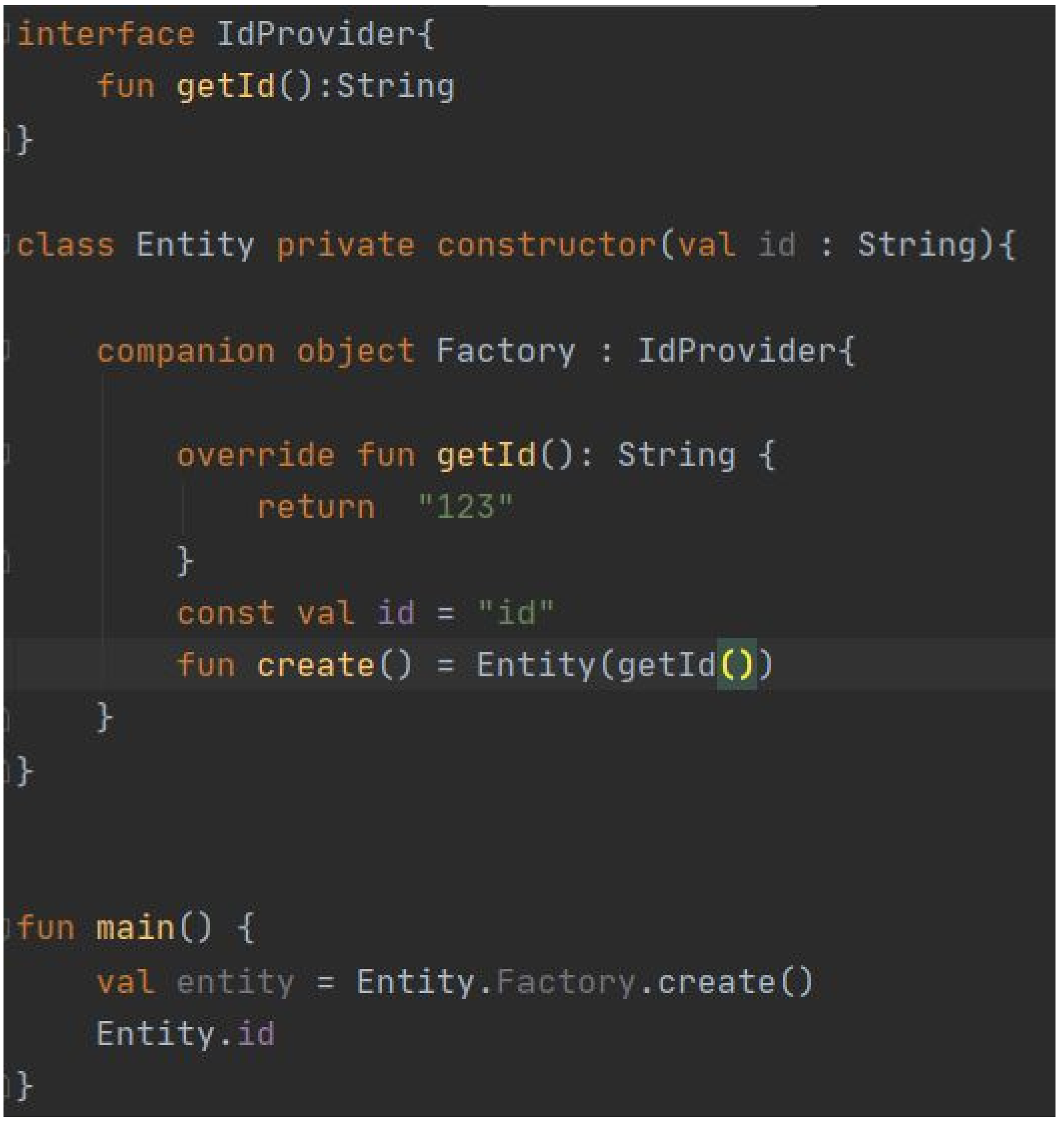
This can be used to create the objects of Entity class. This works because companion objects has access to private properties and functions of that enclosing class.

42. If we have the property id added in the companion object we can reference it form the other calling code as if its a static property.



```kotlin
class Entity private constructor(val id : String){

    companion object Factory{
        const val id = "id"
        fun create() = Entity(id)
    }
}


fun main() {
    val entity = Entity.Factory.create()
    Entity.id
}
```

43. Companion objects are like any other class they can implement any other interface.

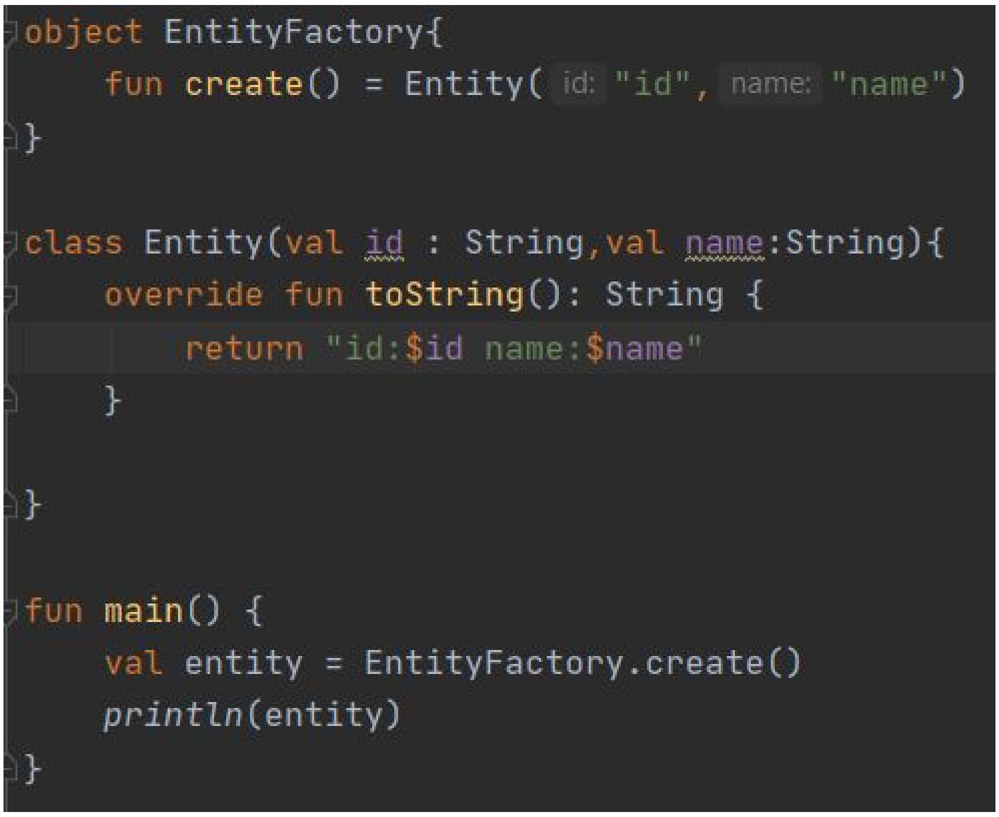


```kotlin
interface IdProvider{
    fun getId():String
}

class Entity private constructor(val id : String){

    companion object Factory : IdProvider{

        override fun getId(): String {
            return  "123"
        }
        const val id = "id"
        fun create() = Entity(getId())
    }
}



fun main() {
    val entity = Entity.Factory.create()
    Entity.id
}
```

In this way the companion objects are flexible. We can use them to compose the other type of behaviors, Store the semi-static properties, or methods and use them to create factories by referencing private inner properties or methods of the enclosing class. This can be leveraged to have static classes members and fields from like JAVA.

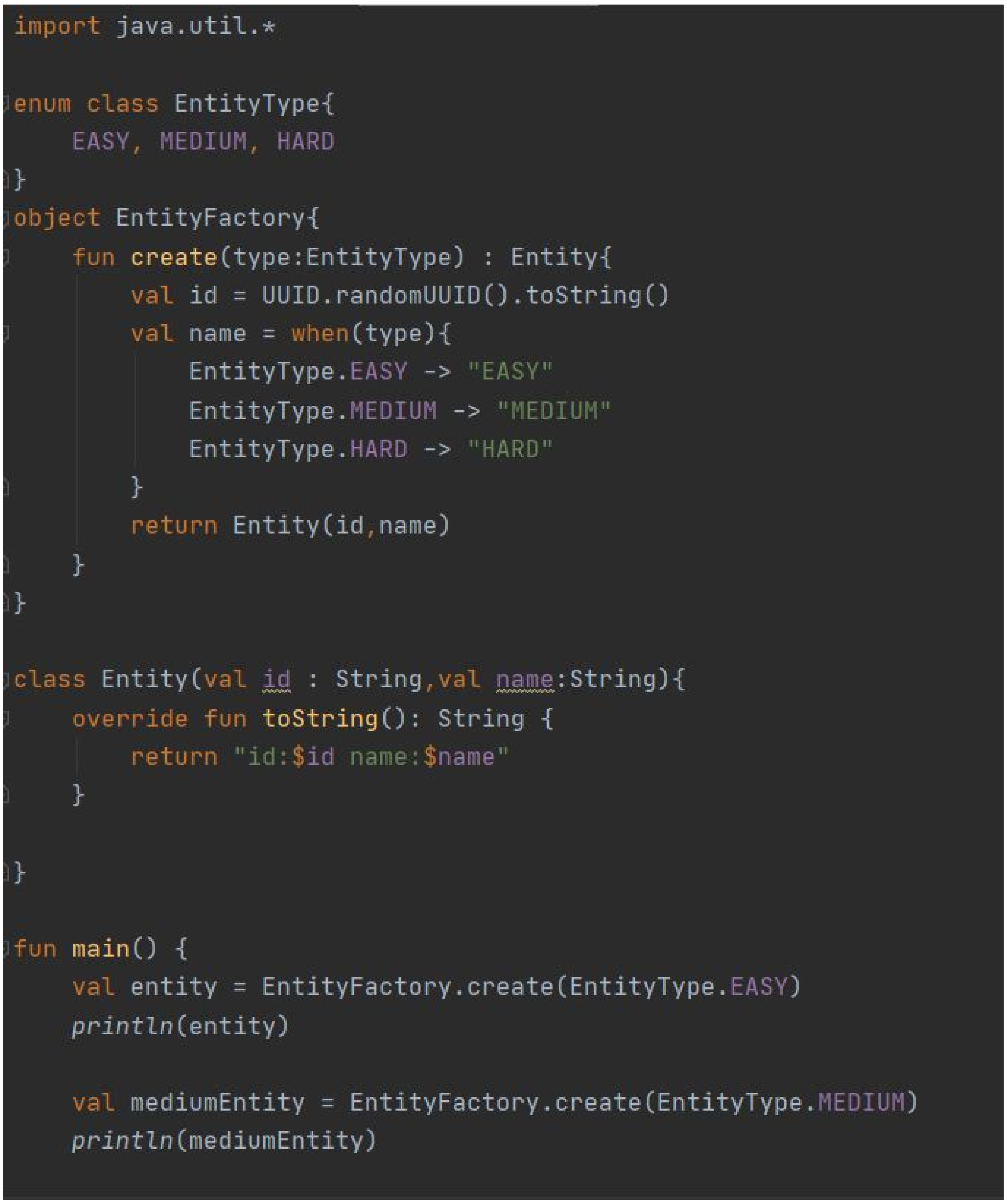
44. Object Declaration in Kotlin

Object declaration is a convenient way of creating thread safe Singletons within Kotlin.

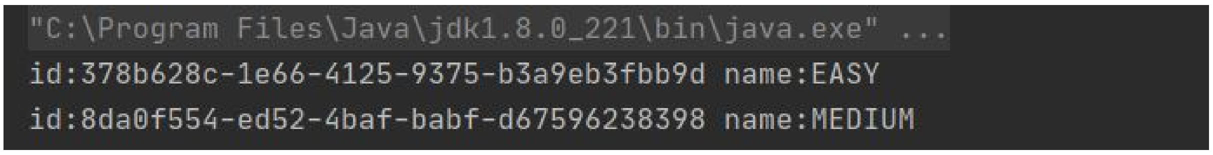


```kotlin
object EntityFactory{
    fun create() = Entity( id: "id", name: "name")
}

class Entity(val id : String,val name:String){
    override fun toString(): String {
        return "id:$id name:$name"
    }

}

fun main() {
    val entity = EntityFactory.create()
    println(entity)
}
```

45.Enum classes in Kotlin

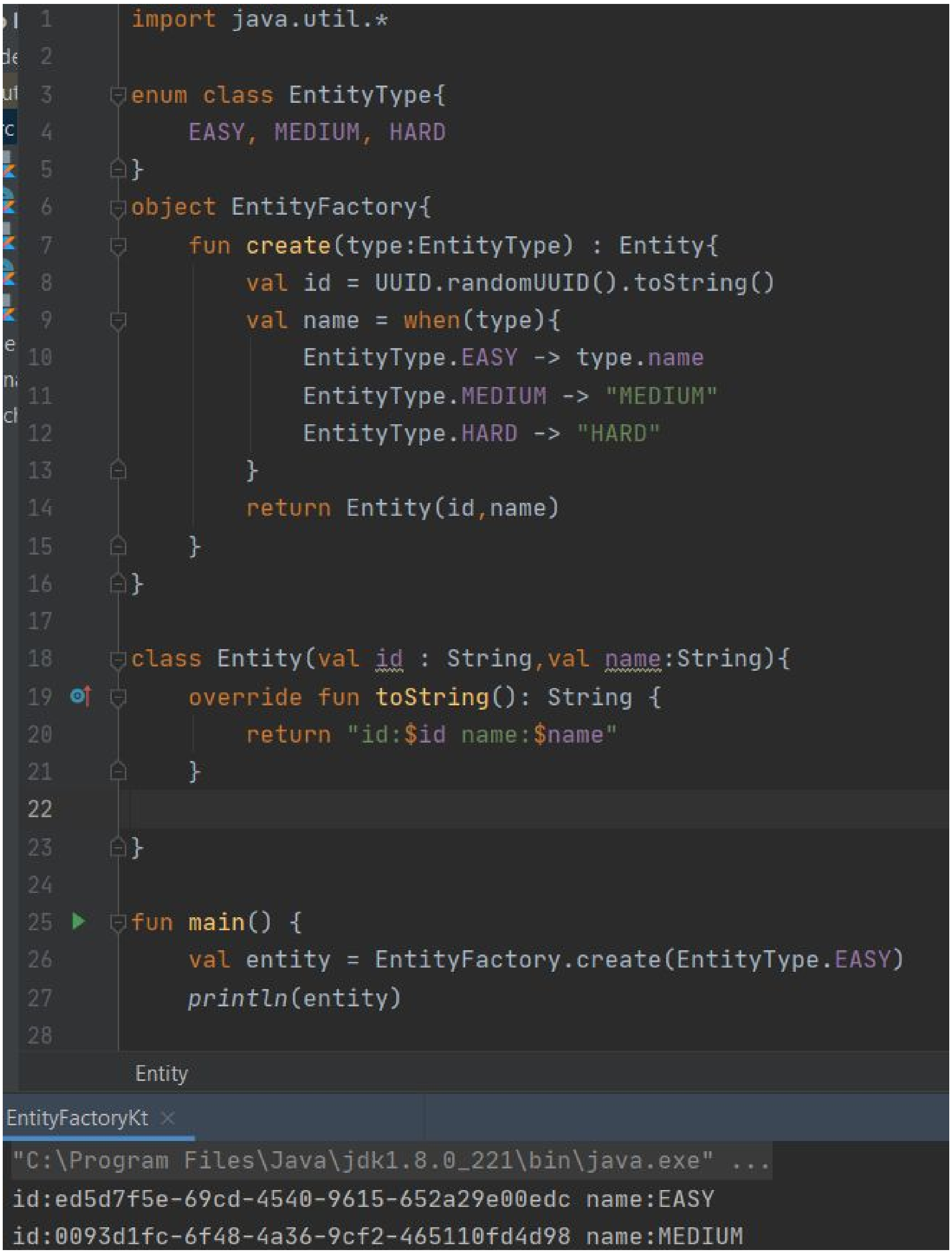


```kotlin
import java.util.*

enum class EntityType{
    EASY, MEDIUM, HARD
}
object EntityFactory{
    fun create(type:EntityType) : Entity{
        val id = UUID.randomUUID().toString()
        val name = when(type){
            EntityType.EASY -> "EASY"
            EntityType.MEDIUM -> "MEDIUM"
            EntityType.HARD -> "HARD"
        }
        return Entity(id,name)
    }
}

class Entity(val id : String,val name:String){
    override fun toString(): String {
        return "id:$id name:$name"
    }

}

fun main() {
    val entity = EntityFactory.create(EntityType.EASY)
    println(entity)

    val mediumEntity = EntityFactory.create(EntityType.MEDIUM)
    println(mediumEntity)
```



```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
id:378b628c-1e66-4125-9375-b3a9eb3fbb9d name:EASY
id:8da0f554-ed52-4baf-babf-d67596238398 name:MEDIUM
```

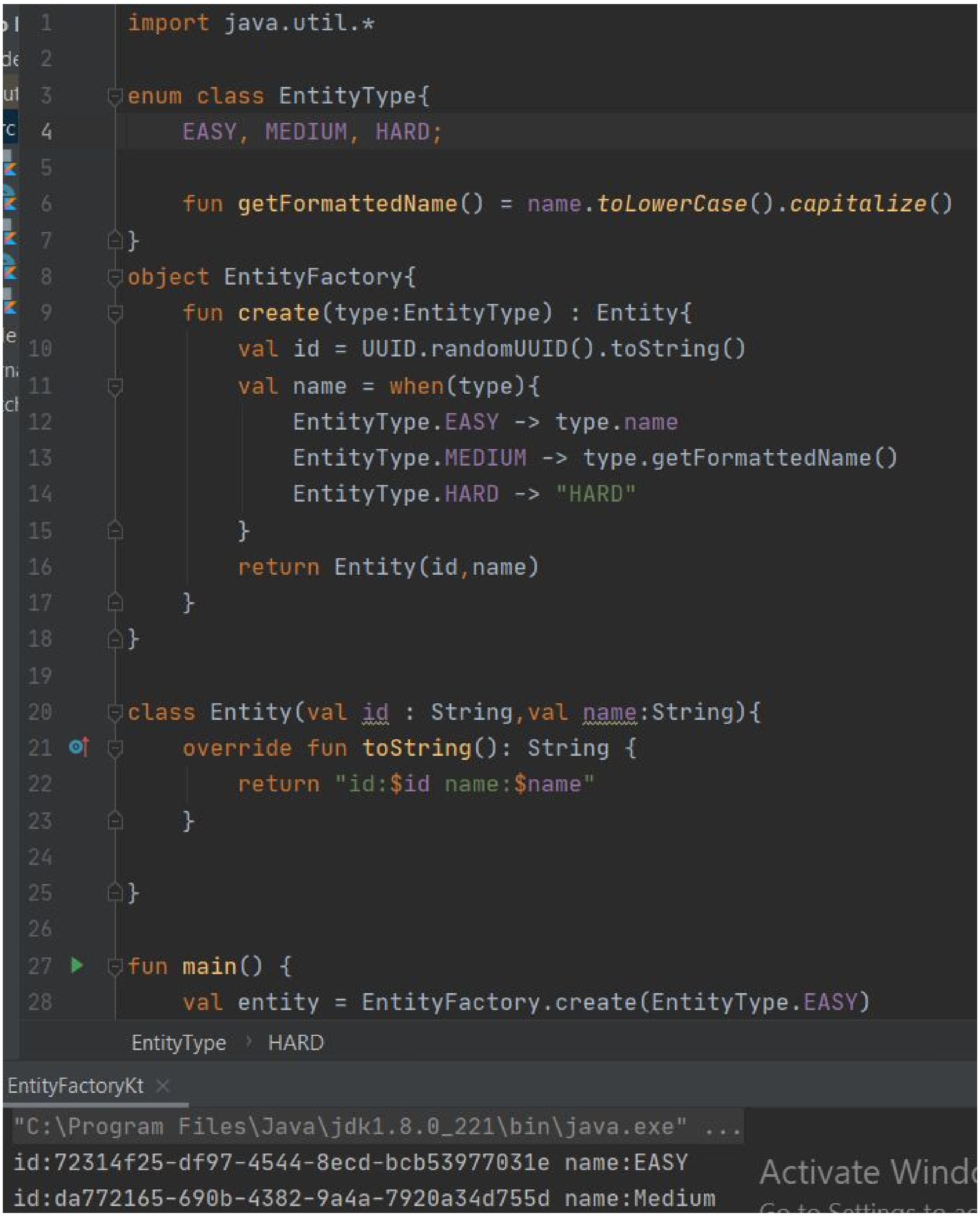
Here we are mapping the Entity type to a name. So here we are mapping the name “EASY” very closely to the name of class itself. We can make it easier and encapsulated.
By taking the advantage of name property on the enum class .



```kotlin
import java.util.*

enum class EntityType{
    EASY, MEDIUM, HARD
}
object EntityFactory{
    fun create(type:EntityType) : Entity{
        val id = UUID.randomUUID().toString()
        val name = when(type){
            EntityType.EASY -> type.name
            EntityType.MEDIUM -> "MEDIUM"
            EntityType.HARD -> "HARD"
        }
        return Entity(id,name)
    }
}

class Entity(val id : String,val name:String){
    override fun toString(): String {
        return "id:$id name:$name"
    }

}

fun main() {
    val entity = EntityFactory.create(EntityType.EASY)
    println(entity)

```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
id:ed5d7f5e-69cd-4540-9615-652a29e00edc name:EASY
id:0093d1fc-6f48-4a36-9cf2-465110fd4d98 name:MEDIUM
```

Still here we don’t have much control over the formatting.

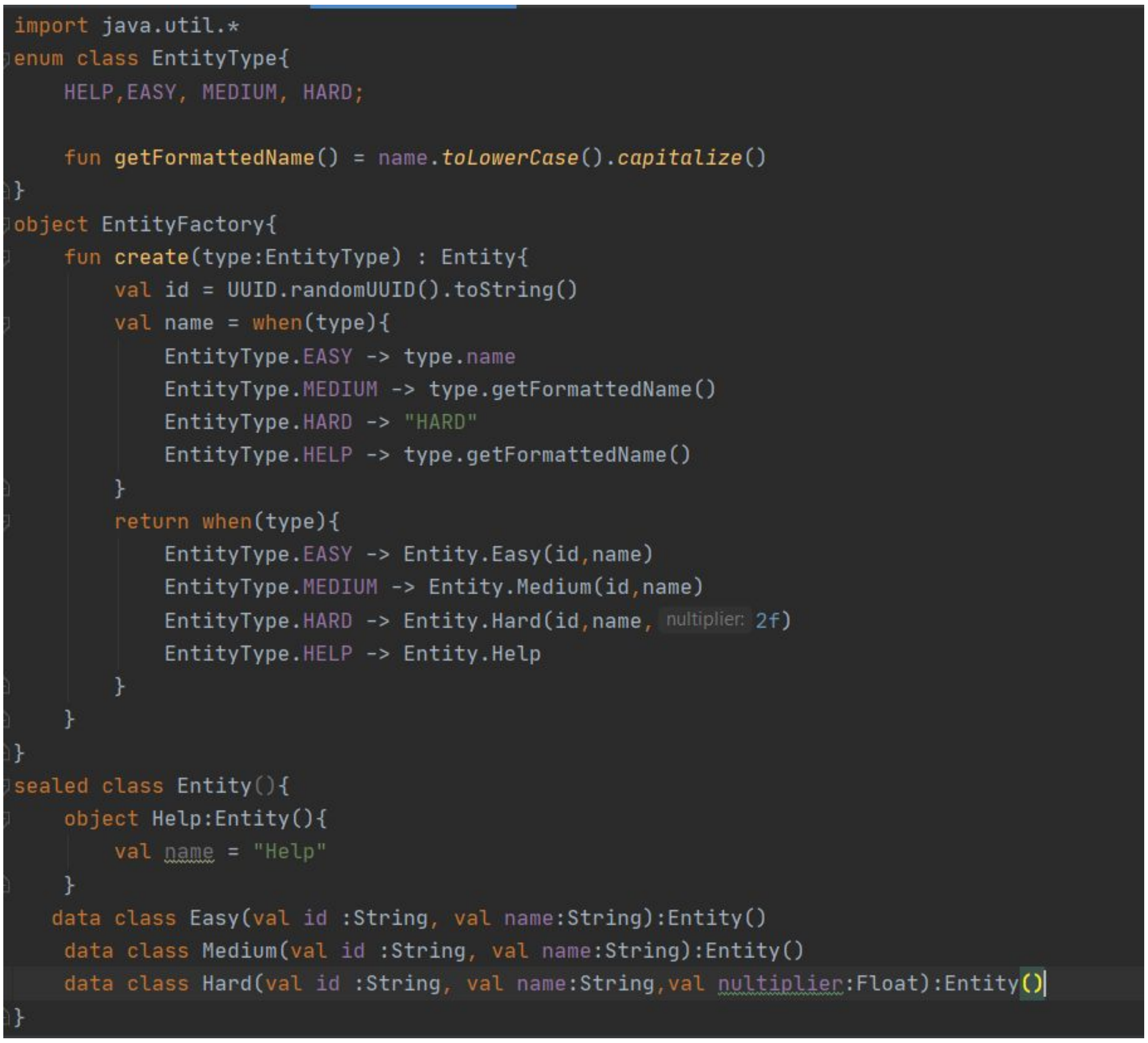


```kotlin
import java.util.*

enum class EntityType{
    EASY, MEDIUM, HARD;

    fun getFormattedName() = name.toLowerCase().capitalize()
}
object EntityFactory{
    fun create(type:EntityType) : Entity{
        val id = UUID.randomUUID().toString()
        val name = when(type){
            EntityType.EASY -> type.name
            EntityType.MEDIUM -> type.getFormattedName()
            EntityType.HARD -> "HARD"
        }
        return Entity(id,name)
    }
}

class Entity(val id : String,val name:String){
    override fun toString(): String {
        return "id:$id name:$name"
    }

}

fun main() {
    val entity = EntityFactory.create(EntityType.EASY)
```

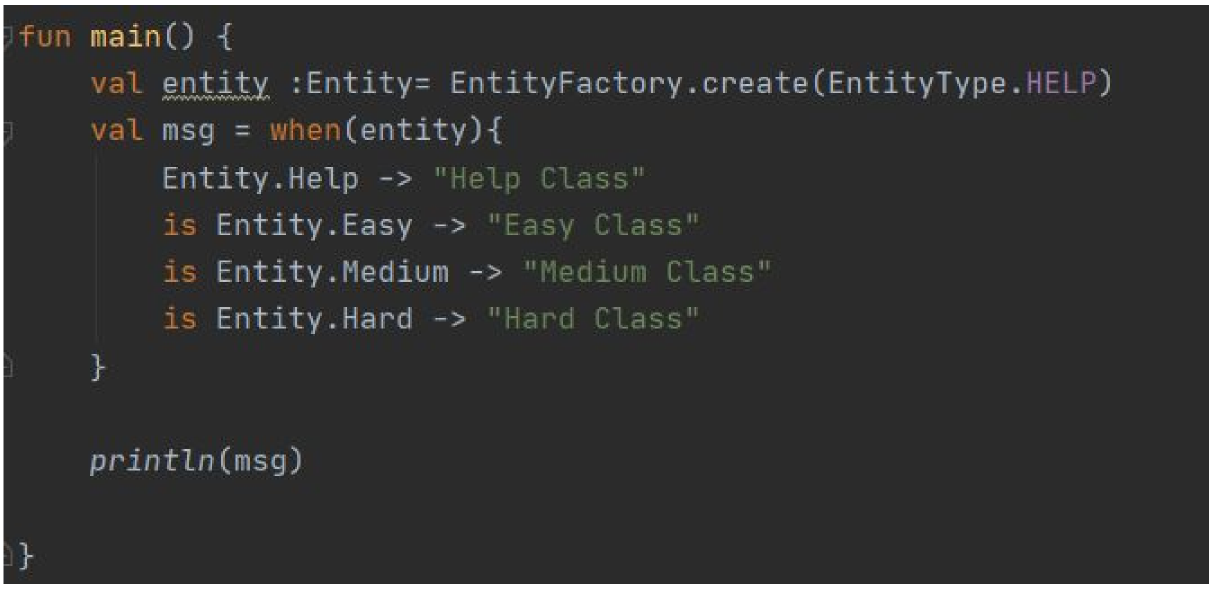
```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
id:72314f25-df97-4544-8ecd-bcb53977031e name:EASY
id:da772165-690b-4382-9a4a-7920a34d755d name:Medium
```

46. Sealed Classes in Kotlin
Sealed classes allows us to define restricted class hierarchies, i.e. we can define the certain number of classes all extending a base class but those class are the ones which can only extend the base class.
In the given screenshot all of the types within the sealed class all inherit from the Entity but have different types of properties, this is one of the key differentiators between sealed classes and enum classes; for the sealed classes you can have different properties and

methods for these types and compiler can form the smart casting to allow you use these different properties and methods as you like. We can also use the different types of classes itself within the sealed class. So you can observe the classes are declared as data classes. We can also use object declaration within sealed class hierarchies.

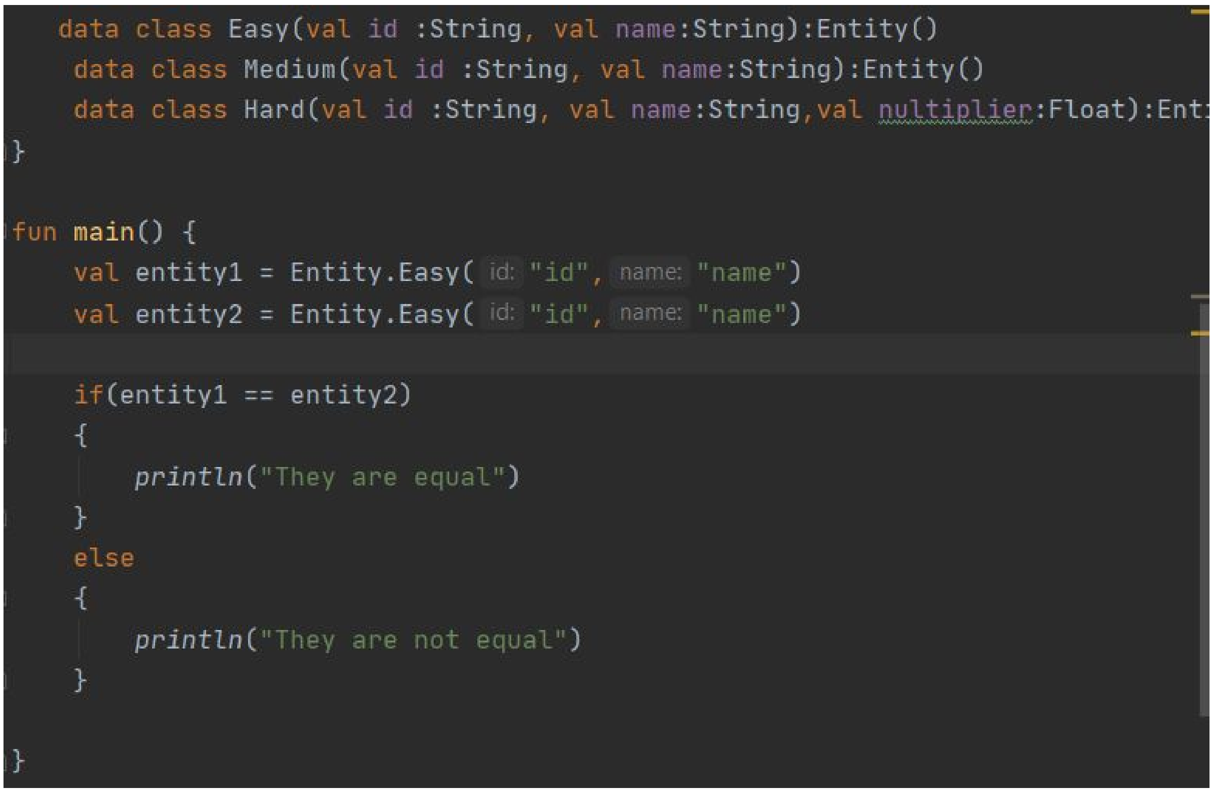


```
import java.util.*
enum class EntityType{
    HELP,EASY, MEDIUM, HARD;

    fun getFormattedName() = name.toLowerCase().capitalize()
}
object EntityFactory{
    fun create(type:EntityType) : Entity{
        val id = UUID.randomUUID().toString()
        val name = when(type){
            EntityType.EASY -> type.name
            EntityType.MEDIUM -> type.getFormattedName()
            EntityType.HARD -> "HARD"
            EntityType.HELP -> type.getFormattedName()
        }
        return when(type){
            EntityType.EASY -> Entity.Easy(id,name)
            EntityType.MEDIUM -> Entity.Medium(id,name)
            EntityType.HARD -> Entity.Hard(id,name, nultiplier: 2f)
            EntityType.HELP -> Entity.Help
        }
    }
}
sealed class Entity(){
    object Help:Entity(){
        val name = "Help"
    }
   data class Easy(val id :String, val name:String):Entity()
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Entity()
}
```



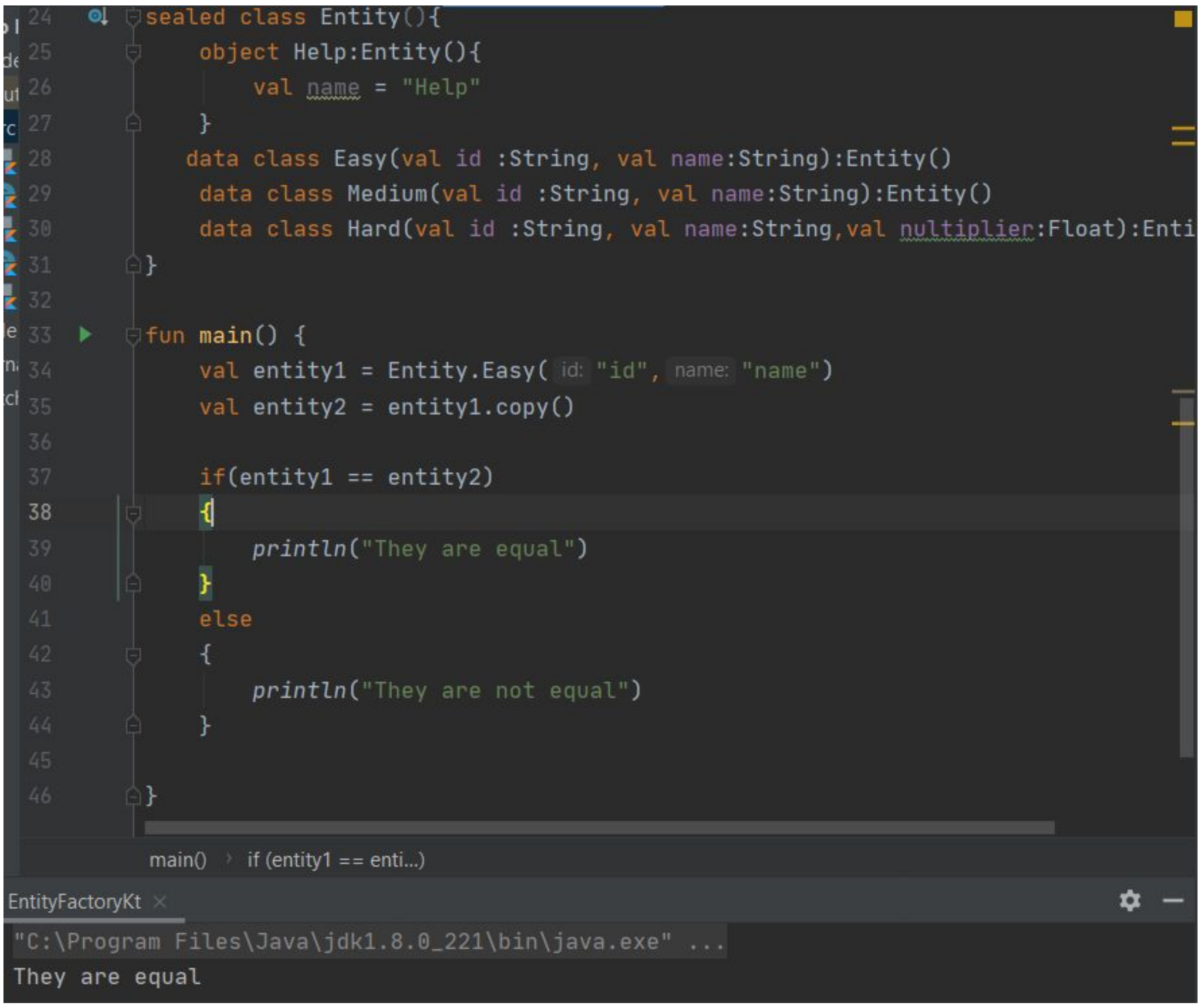
```
fun main() {
    val entity :Entity= EntityFactory.create(EntityType.HELP)
    val msg = when(entity){
        Entity.Help -> "Help Class"
        is Entity.Easy -> "Easy Class"
        is Entity.Medium -> "Medium Class"
        is Entity.Hard -> "Hard Class"
    }

    println(msg)

}
```

47. Data classes in Kotlin :
Data classes are kotlin’s way of providing very concise immutable data types. By defining a class as a data class it means it is going to generate methods such as equals, hashcode, toString automatically for you. It allows you to do the quality comparison on instances of these data classes and treat them as equal(==) if the data they contain is equal.



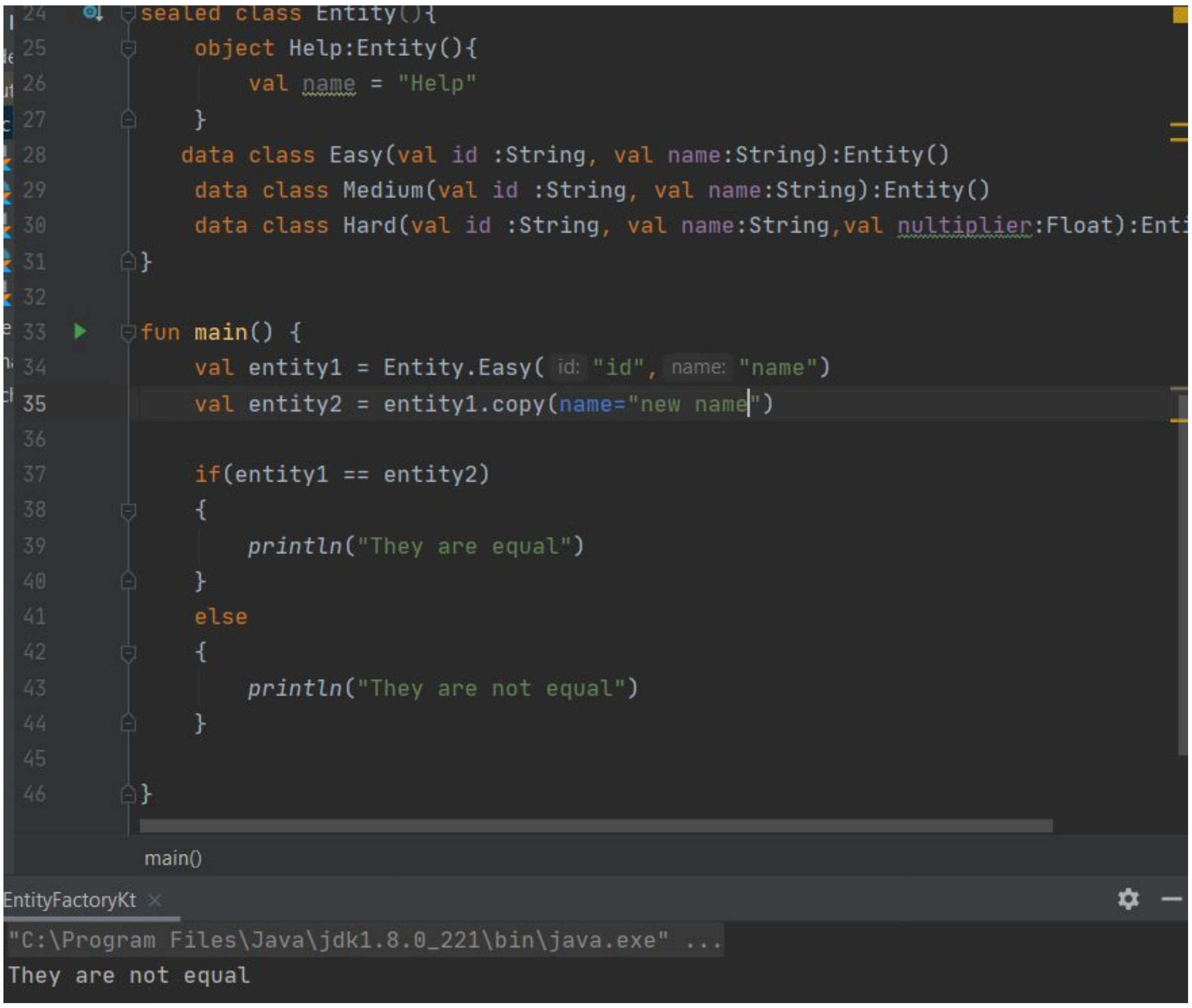
```
   data class Easy(val id :String, val name:String):Entity()
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Ent
}

fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = Entity.Easy( id: "id", name: "name")

    if(entity1 == entity2)
    {
        println("They are equal")
    }
    else
    {
        println("They are not equal")
    }

}
```

This allows us to represent the data from our applications and compares it no matter where it comes from. If all the values are the same then evaluated as True else False.
Also data classes copy constructors.



```
sealed class Entity(){
    object Help:Entity(){
        val name = "Help"
    }
   data class Easy(val id :String, val name:String):Entity()
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Enti
}

fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = entity1.copy()

    if(entity1 == entity2)
    {
        println("They are equal")
    }
    else
    {
        println("They are not equal")
    }

}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
They are equal
```



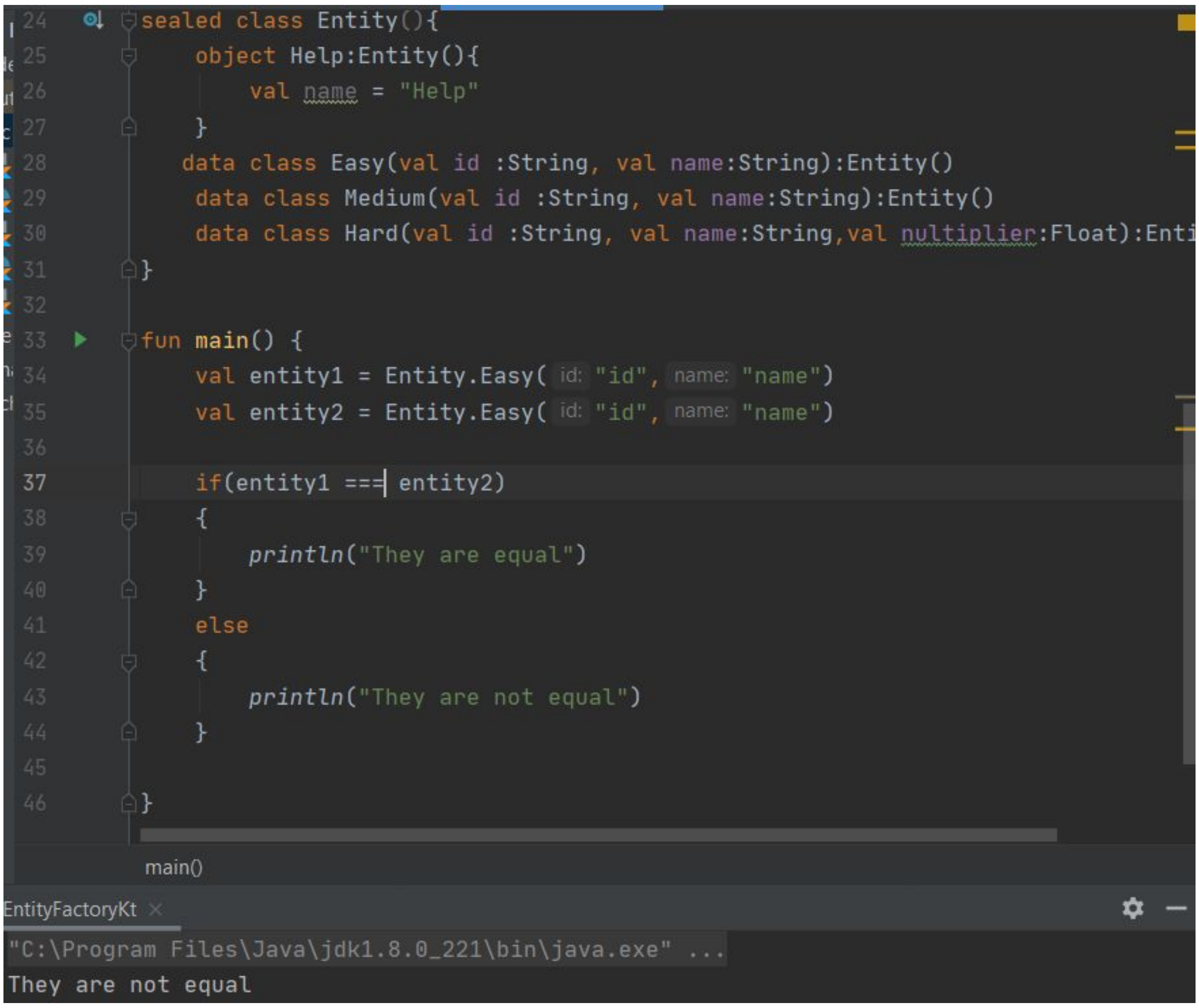
```kotlin
sealed class Entity(){
    object Help:Entity(){
        val name = "Help"
    }
   data class Easy(val id :String, val name:String):Entity()
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Enti
}

fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = entity1.copy(name="new name")

    if(entity1 == entity2)
    {
        println("They are equal")
    }
    else
    {
        println("They are not equal")
    }

}
```

main()

EntityFactoryKt

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
They are not equal
```

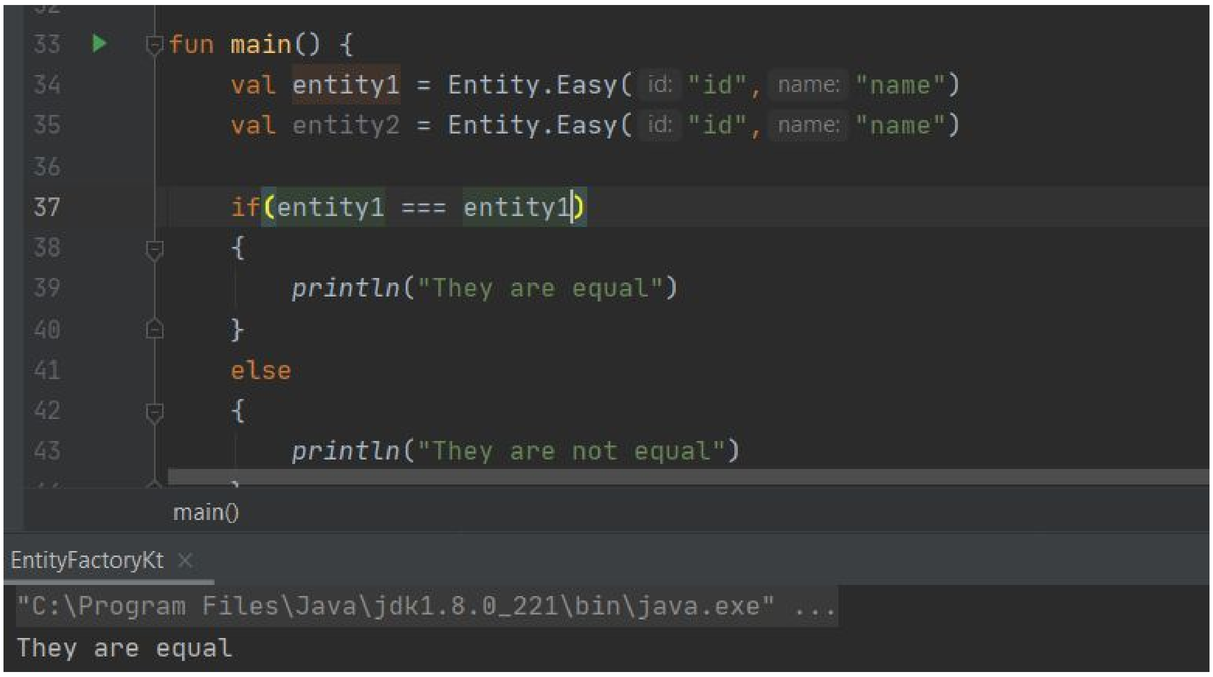
For Referential comparison : ===



```kotlin
sealed class Entity(){
    object Help:Entity(){
        val name = "Help"
    }
   data class Easy(val id :String, val name:String):Entity()
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Enti
}

fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = Entity.Easy( id: "id", name: "name")

    if(entity1 === entity2)
    {
        println("They are equal")
    }
    else
    {
        println("They are not equal")
    }

}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
They are not equal
```

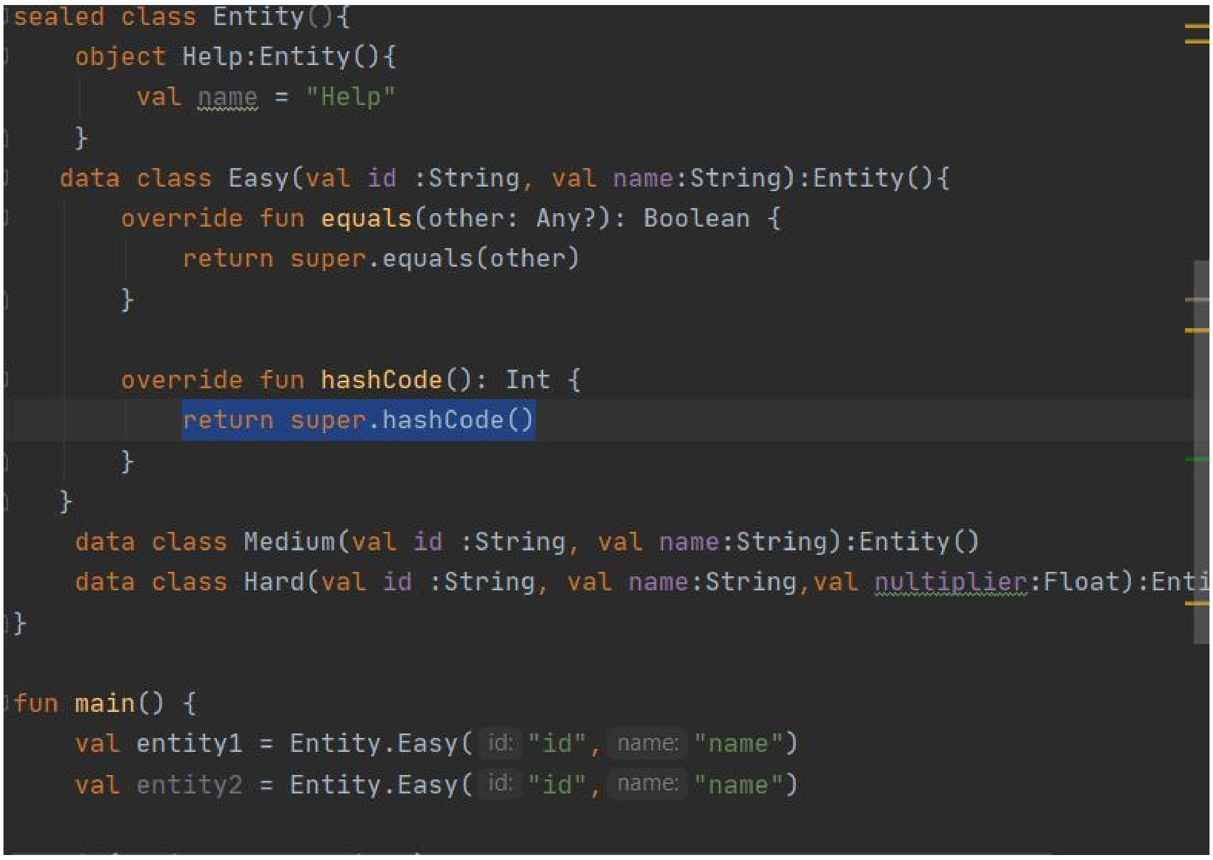
Because not the exact same reference of the objects.



```kotlin
fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = Entity.Easy( id: "id", name: "name")

    if(entity1 === entity1)
    {
        println("They are equal")
    }
    else
    {
        println("They are not equal")
```

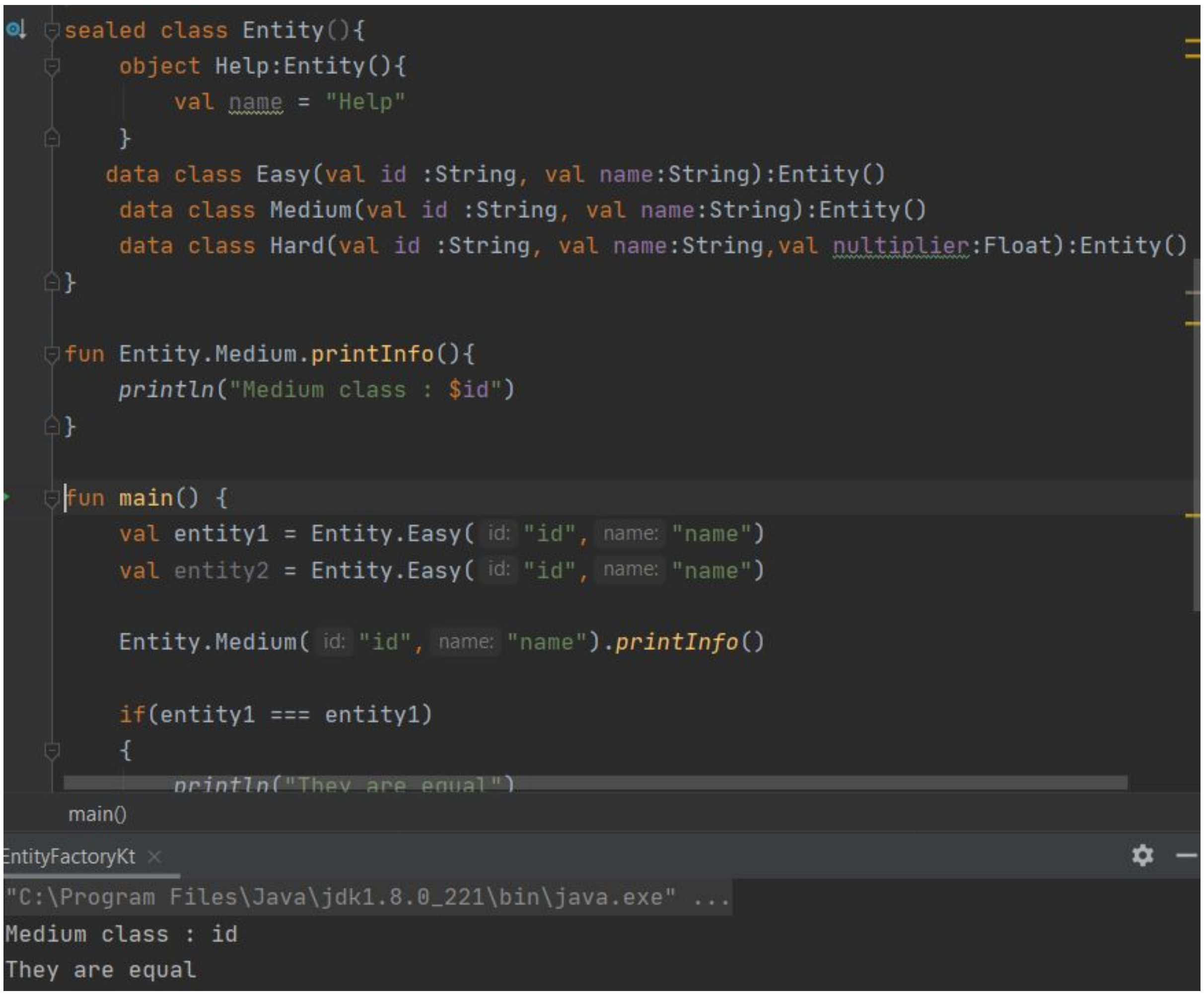
```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
They are equal
```

Same as JAVA these equality comparisons working on the generated equals and hashcode methods generated by the compiler when indicating the data class. However we can update these to change how the equals and hashcode is evaluated. It can be done in following manner:



```
sealed class Entity(){
    object Help:Entity(){
        val name = "Help"
    }
   data class Easy(val id :String, val name:String):Entity(){
       override fun equals(other: Any?): Boolean {
           return super.equals(other)
       }

       override fun hashCode(): Int {
           return super.hashCode()
       }
   }
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Enti
}

fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = Entity.Easy( id: "id", name: "name")
```

48. Extension Function / Properties on an existing class
This is useful in the cases when you cant control the classes but would like to modify the way they are used. You can use your own properties and methods and define a kind of new API around the existing class.



```kotlin
sealed class Entity(){
    object Help:Entity(){
        val name = "Help"
    }
   data class Easy(val id :String, val name:String):Entity()
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Entity()
}

fun Entity.Medium.printInfo(){
    println("Medium class : $id")
}

fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = Entity.Easy( id: "id", name: "name")

    Entity.Medium( id: "id", name: "name").printInfo()

    if(entity1 === entity1)
    {
        println("They are equal")
```

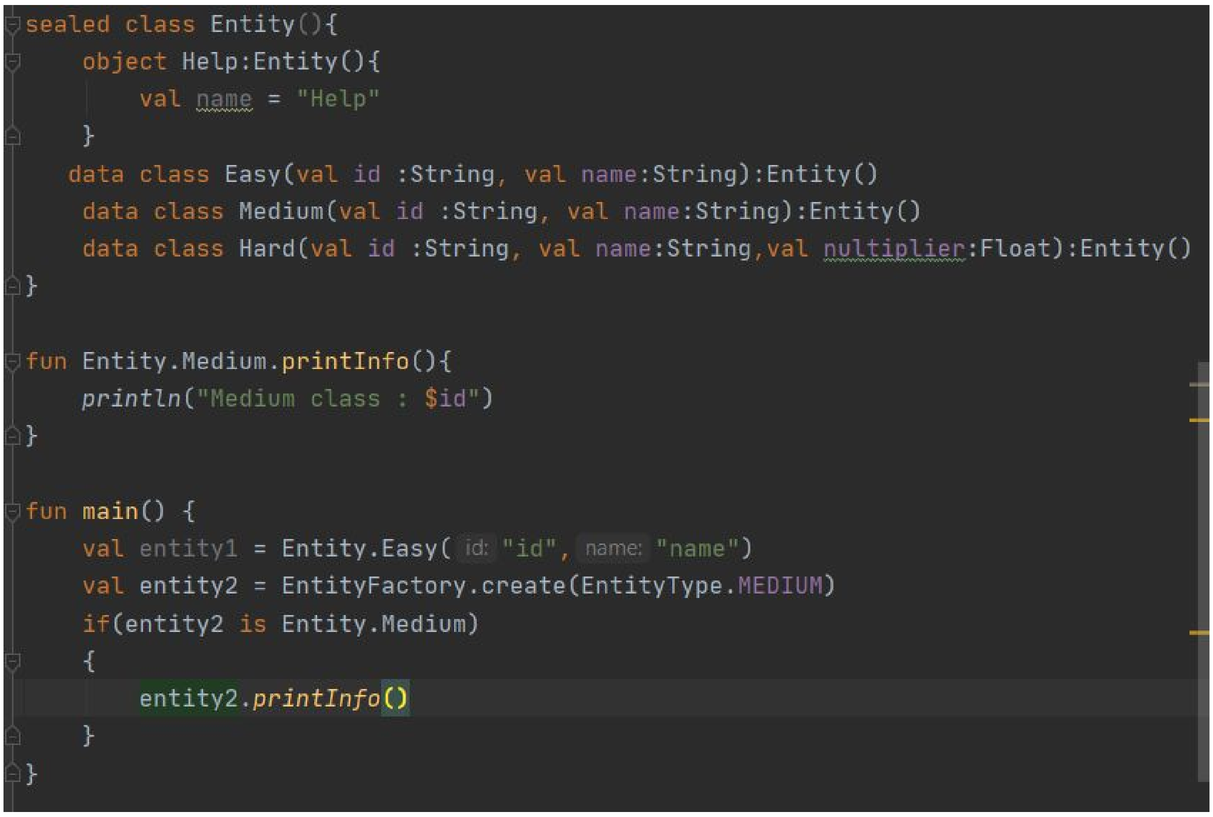
main()

EntityFactoryKt

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Medium class : id
They are equal
```

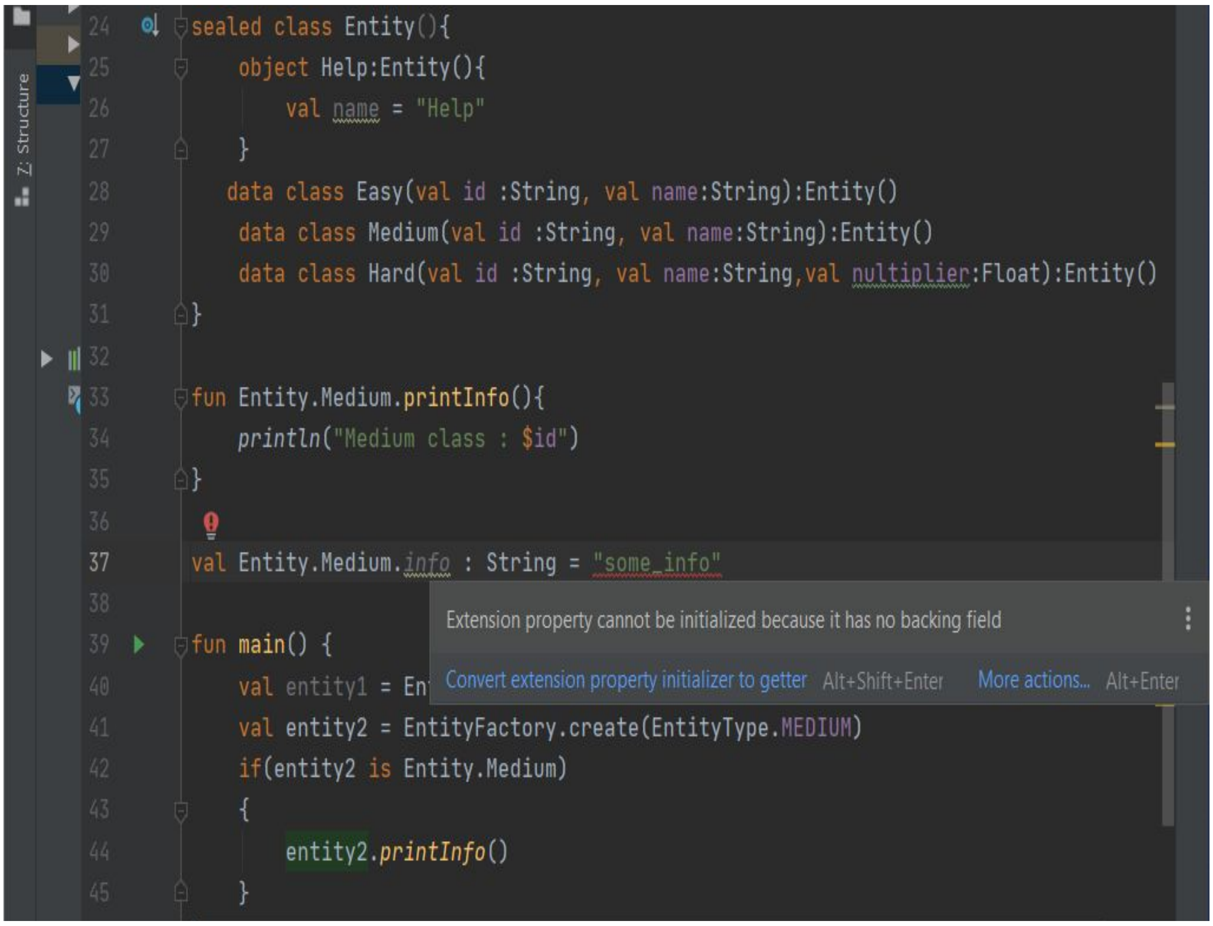
This works if we know the exact type that we are working with.

49. In cases where we dont know the direct type we can work with smart casting.



```kotlin
sealed class Entity(){
    object Help:Entity(){
        val name = "Help"
    }
   data class Easy(val id :String, val name:String):Entity()
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Entity()
}

fun Entity.Medium.printInfo(){
    println("Medium class : $id")
}

fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = EntityFactory.create(EntityType.MEDIUM)
    if(entity2 is Entity.Medium)
    {
        entity2.printInfo()
    }
}
```

50. Issue with the Extension Property

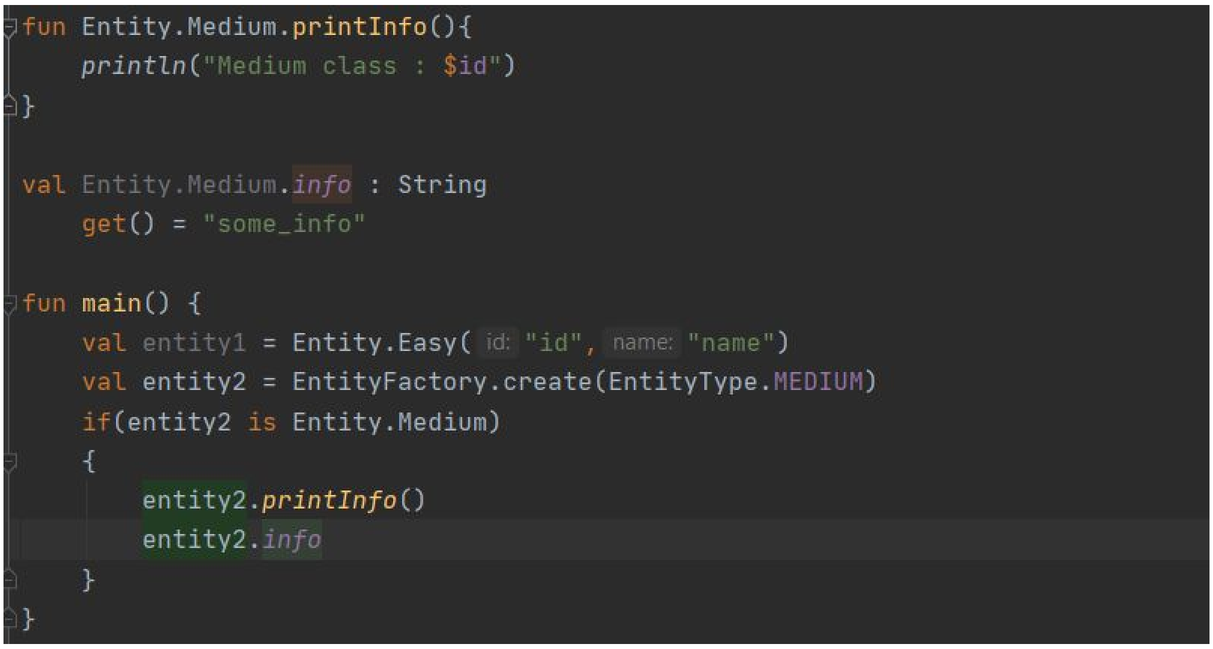


```kotlin
sealed class Entity(){
    object Help:Entity(){
        val name = "Help"
    }
   data class Easy(val id :String, val name:String):Entity()
    data class Medium(val id :String, val name:String):Entity()
    data class Hard(val id :String, val name:String,val nultiplier:Float):Entity()
}

fun Entity.Medium.printInfo(){
    println("Medium class : $id")
}

val Entity.Medium.info : String = "some_info"

fun main() {
    val entity1 = En
    val entity2 = EntityFactory.create(EntityType.MEDIUM)
    if(entity2 is Entity.Medium)
    {
        entity2.printInfo()
    }
```

Extension property cannot be initialized because it has no backing field

Z: Structure
Convert extension property initializer to getter Alt+Shift+Enter
More actions... Alt+Enter

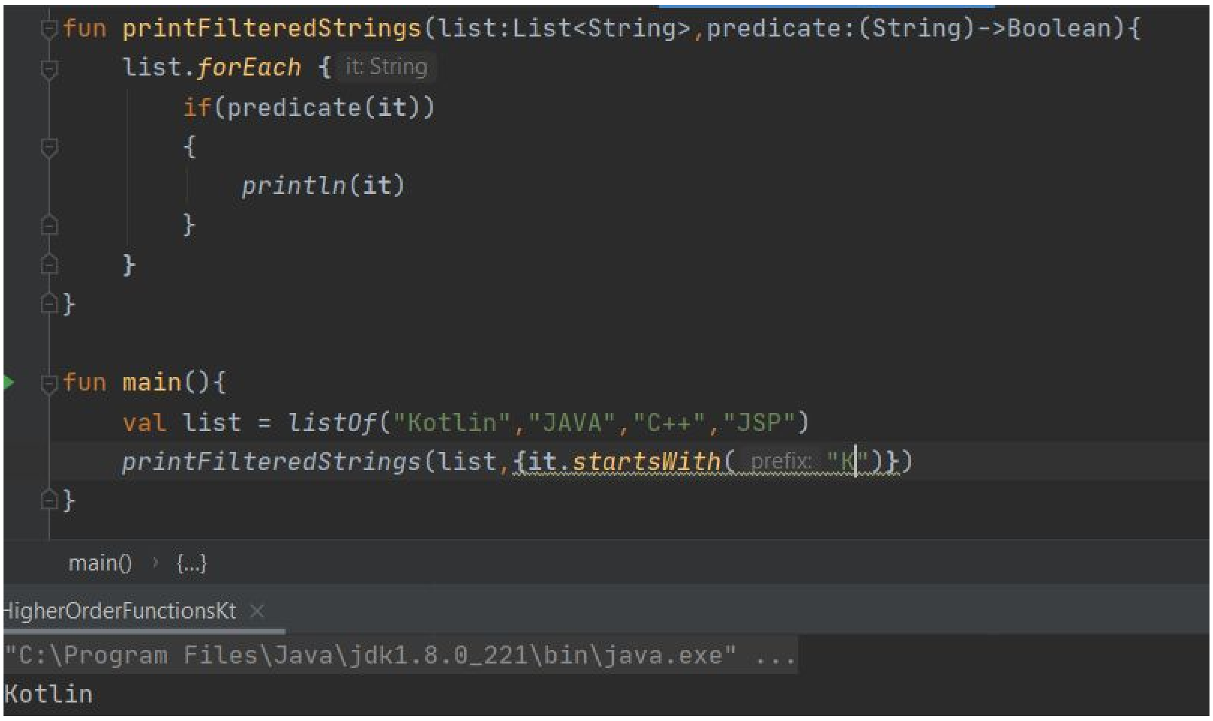
51.



```kotlin
fun Entity.Medium.printInfo(){
    println("Medium class : $id")
}

val Entity.Medium.info : String
    get() = "some_info"

fun main() {
    val entity1 = Entity.Easy( id: "id", name: "name")
    val entity2 = EntityFactory.create(EntityType.MEDIUM)
    if(entity2 is Entity.Medium)
    {
        entity2.printInfo()
        entity2.info
    }
}
```

Within Kotlin standard library many functions and operations work by using extension functions and classes are effective when using templated types. Because it allows to define the common functionality across any type that matches that template.

52. Advanced Functions in Kotlin
Higher Order Functions : The function that either returns another function or they take functions as the parameter values. Kotlin's standard library is built on top of higher order functions, allowing us to write highly functional code using those standard libraries.



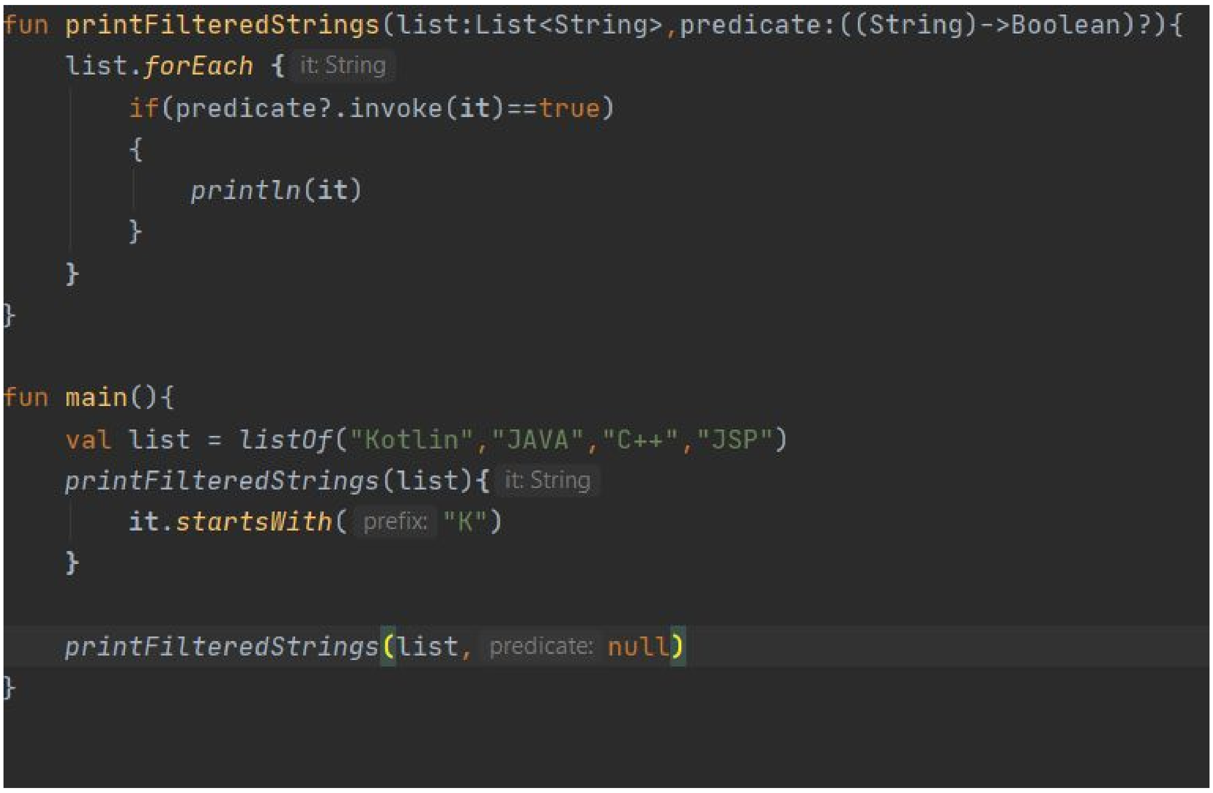
```kotlin
fun printFilteredStrings(list:List<String>,predicate:(String)->Boolean){
    list.forEach { it: String
        if(predicate(it))
        {
            println(it)
        }
    }
}

fun main(){
    val list = listOf("Kotlin","JAVA","C++","JSP")
    printFilteredStrings(list,{it.startsWith( prefix: "K")})
}
```

main() › {...}

HigherOrderFunctionsKt ×

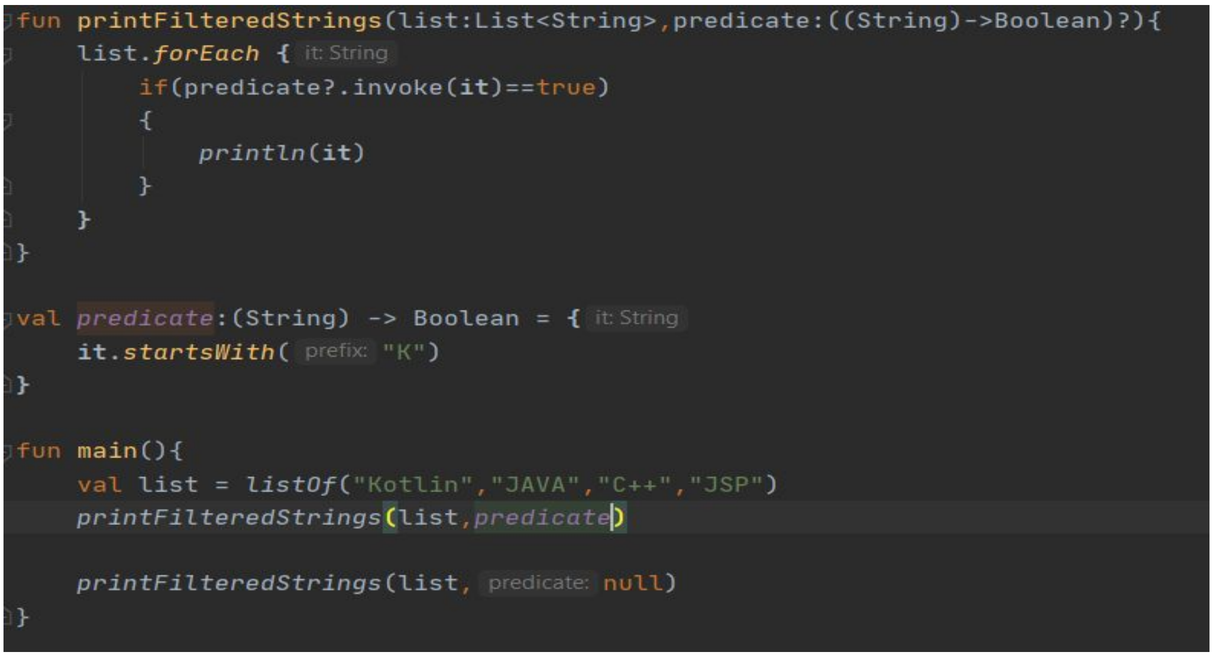
```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Kotlin
```

53. If the last parameter of a function is a function then you can specify that as a Lambda outside the function body. In the above code we are are calling the function parameter directly as if its a regular function. If we wanted to make the function type in the higher order function declaration nullable then we see an error in the implementation that is the problem due to nullability so we solve this using invoke() on that functional type. (Safe invoke calling).



```
fun printFilteredStrings(list:List<String>,predicate:((String)->Boolean)?){
    list.forEach { it: String
        if(predicate?.invoke(it)==true)
        {
            println(it)
        }
    }
}

fun main(){
    val list = listOf("Kotlin","JAVA","C++","JSP")
    printFilteredStrings(list){ it: String
        it.startsWith( prefix: "K")
    }

    printFilteredStrings(list, predicate: null)
}
```

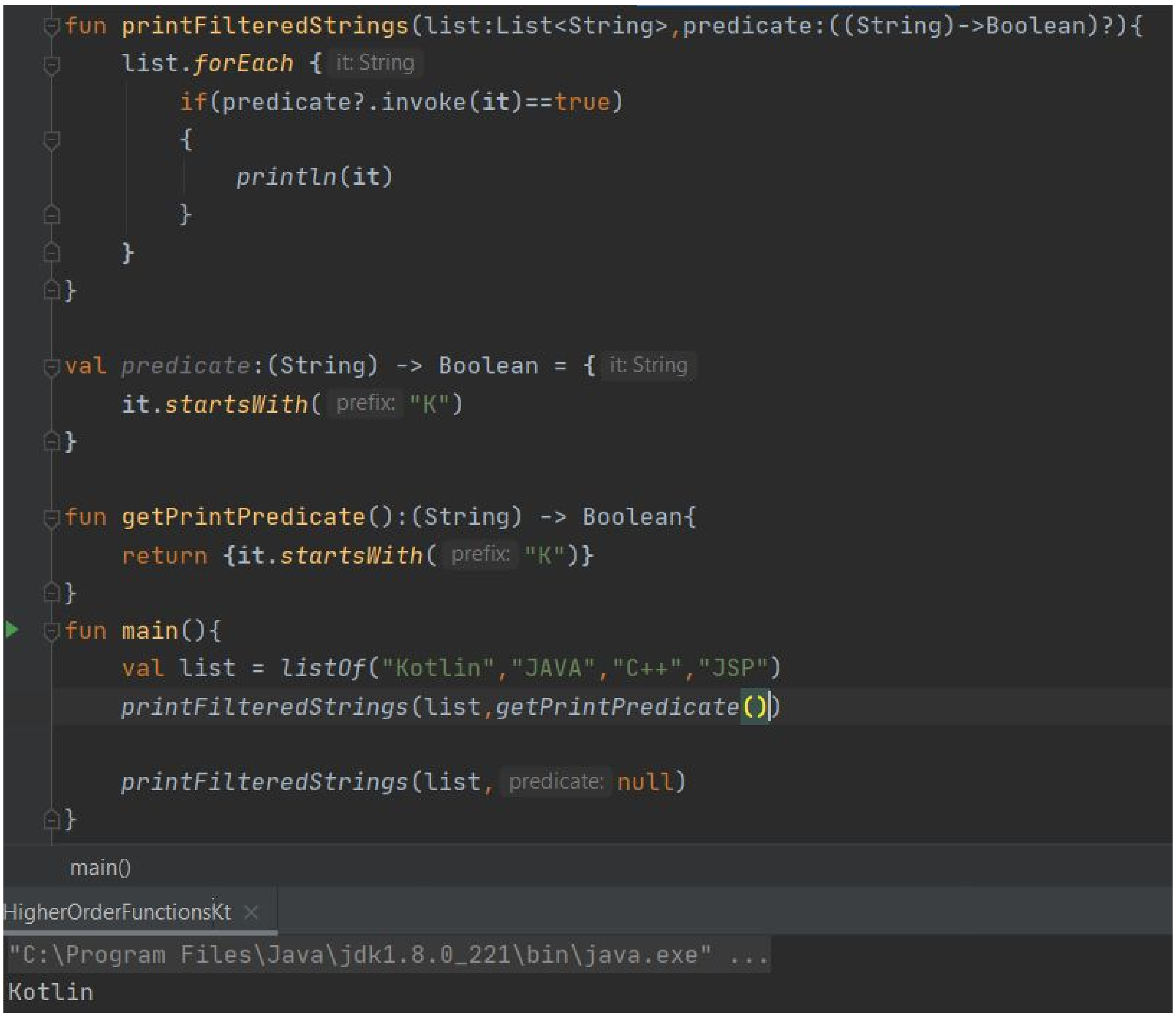
In Kotlin the functions work as the types.
Kotlin as the idea called functional types that means we can define a variable of functional type and can pass that variable in any time we need a function parameter that match that function signature.



```
fun printFilteredStrings(list:List<String>,predicate:((String)->Boolean)?){
    list.forEach { it: String
        if(predicate?.invoke(it)==true)
        {
            println(it)
        }
    }
}

val predicate:(String) -> Boolean = { it: String
    it.startsWith( prefix: "K")
}

fun main(){
    val list = listOf("Kotlin","JAVA","C++","JSP")
    printFilteredStrings(list,predicate)

    printFilteredStrings(list, predicate: null)
}
```

This allows us to store functions as variables. This could be useful where we need optional input handling. For a view on some screen and you want to be able to specify a clickListener For that view; you could define that as a lambda property on some class and allow client code to set that clickListener as needed. As mentioned before the higher order functions include functions which take other functions as parameters as well as the functions that return other functions.

54. We can call the getPrintPredicate() function which then return a function that can be used as the predicate for the printFilteredStrings()



```
fun printFilteredStrings(list:List<String>,predicate:((String)->Boolean)?){
    list.forEach { it: String
        if(predicate?.invoke(it)==true)
        {
            println(it)
        }
    }
}

val predicate:(String) -> Boolean = { it: String
    it.startsWith( prefix: "K")
}

fun getPrintPredicate():(String) -> Boolean{
    return {it.startsWith( prefix: "K")}
}
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JSP")
    printFilteredStrings(list,getPrintPredicate())

    printFilteredStrings(list, predicate: null)
}
```

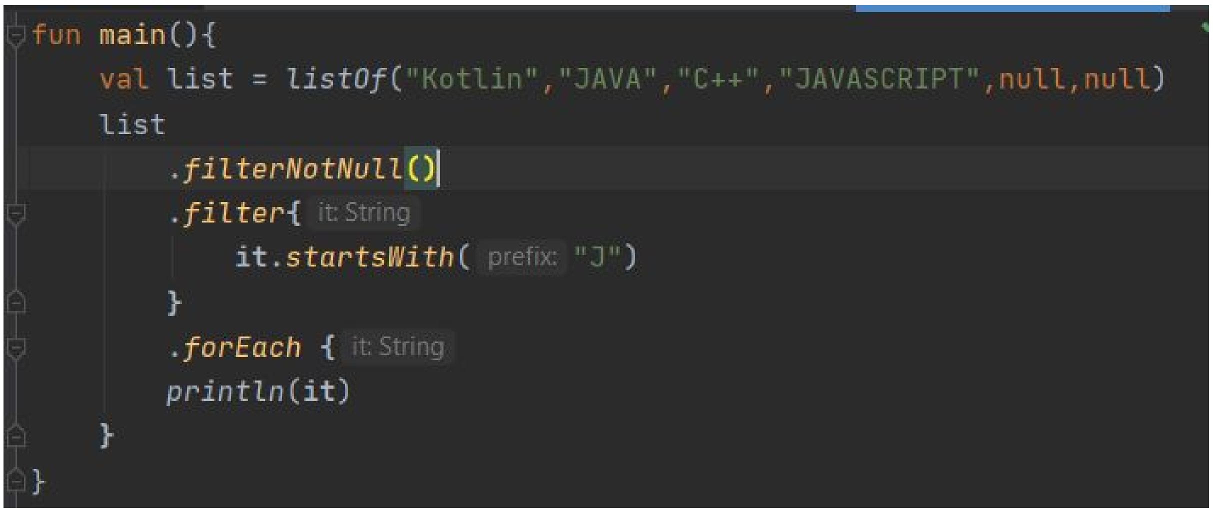
main()

HigherOrderFunctionsKt

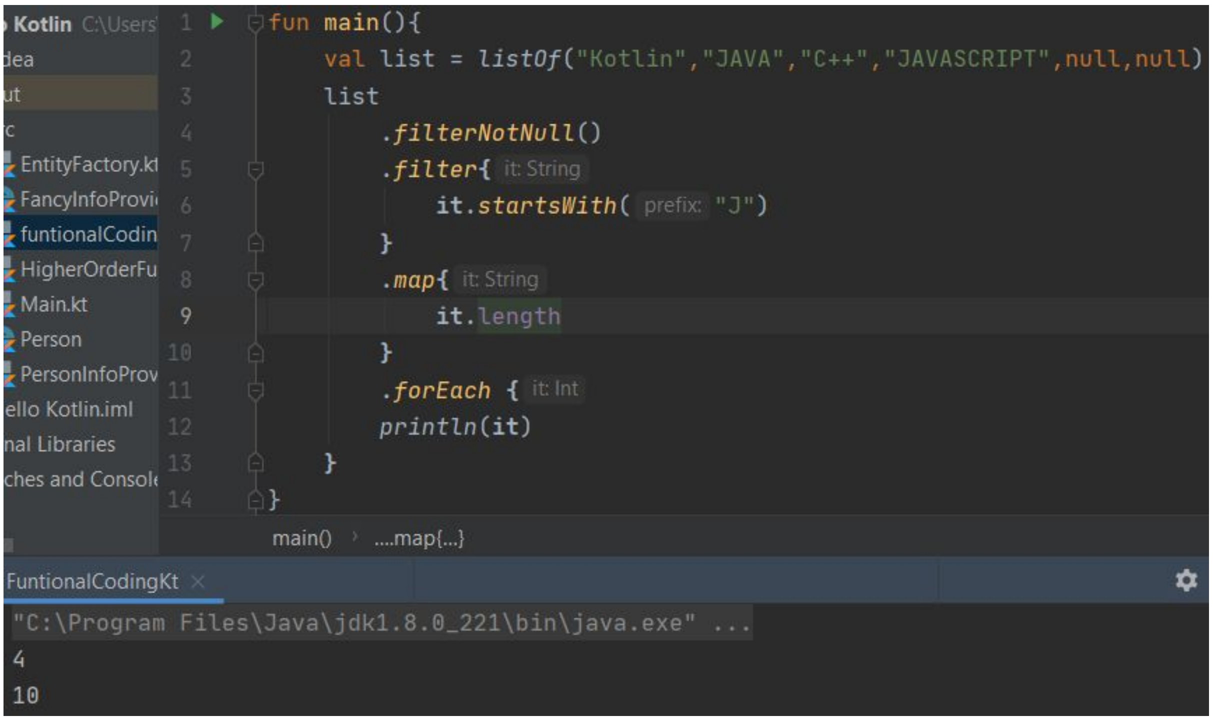
```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Kotlin
```

So Higher Order Functions can be used as both inputs and outputs.

55. If we look at implementation of forEach; as we noticed that it is an extension function as well as the higher order function. So, forEarch works on generic iterable type and takes in a function parameter; that takes in that generic type and returns Unit.
The power of Generic Type, extension function and higher order function allows us to write single implementations of these methods and reuse them any type that we can think off. This allows us to write more functional codes without having to redefine these methods and functions for all the different types.

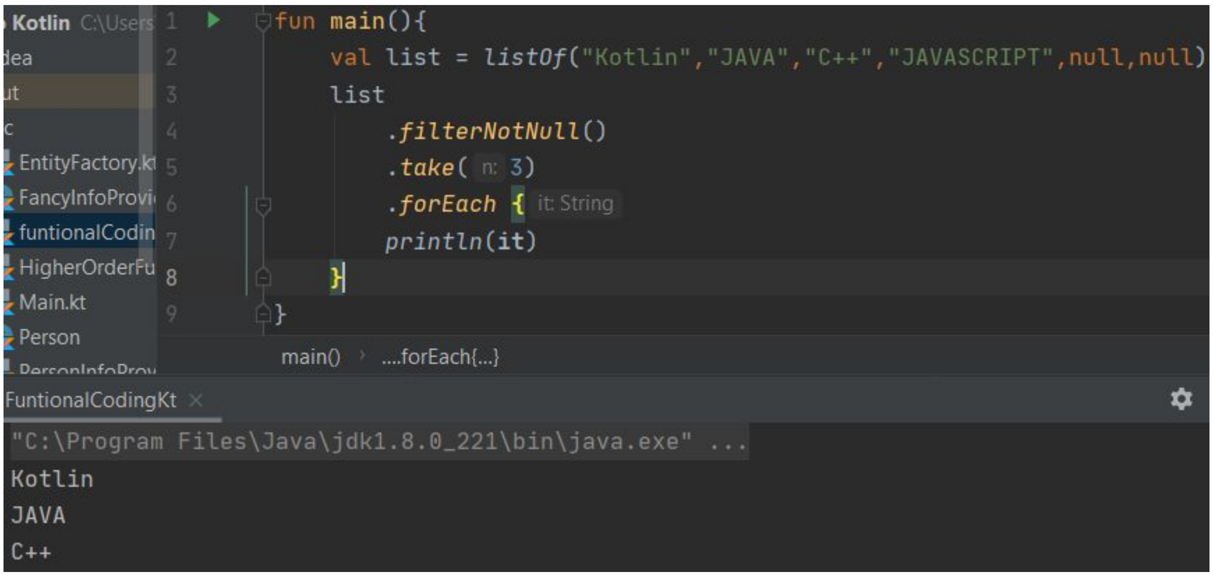


```kotlin
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    list
        .filterNotNull()
        .filter{ it: String
            it.startsWith( prefix: "J")
        }
        .forEach { it: String
        println(it)
    }
}
```



```kotlin
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    list
        .filterNotNull()
        .filter{ it: String
            it.startsWith( prefix: "J")
        }
        .map{ it: String
            it.length
        }
        .forEach { it: Int
        println(it)
    }
}
```

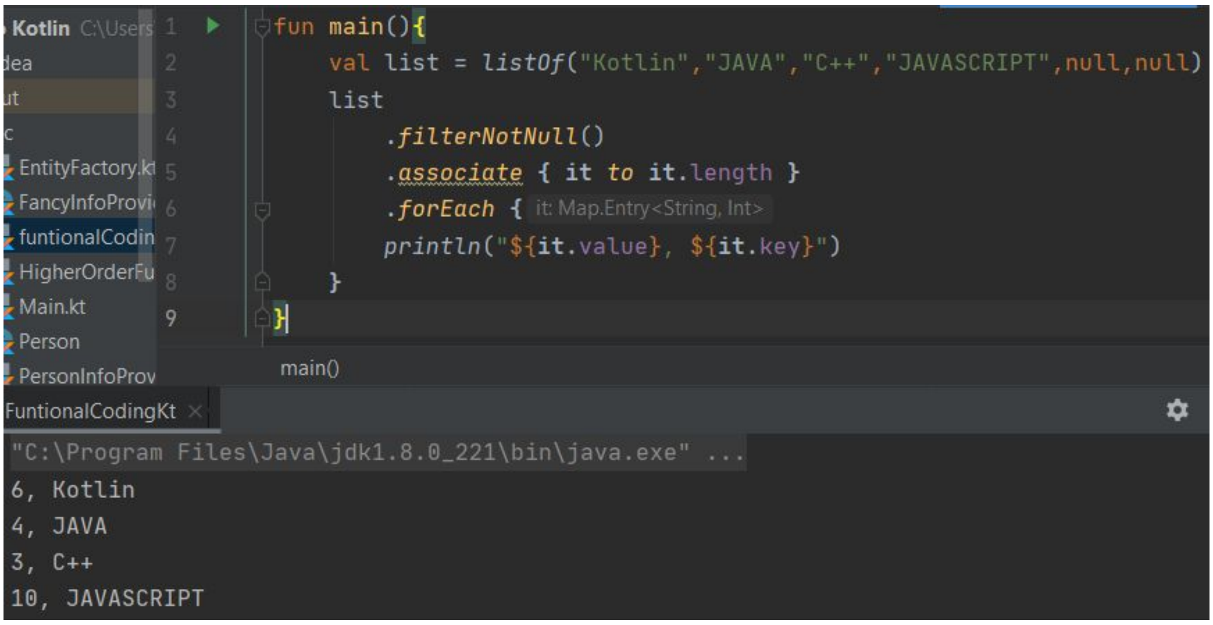
```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
4
10
```



```kotlin
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    list
        .filterNotNull()
        .take( n: 3)
        .forEach { it: String
        println(it)
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Kotlin
JAVA
C++
```

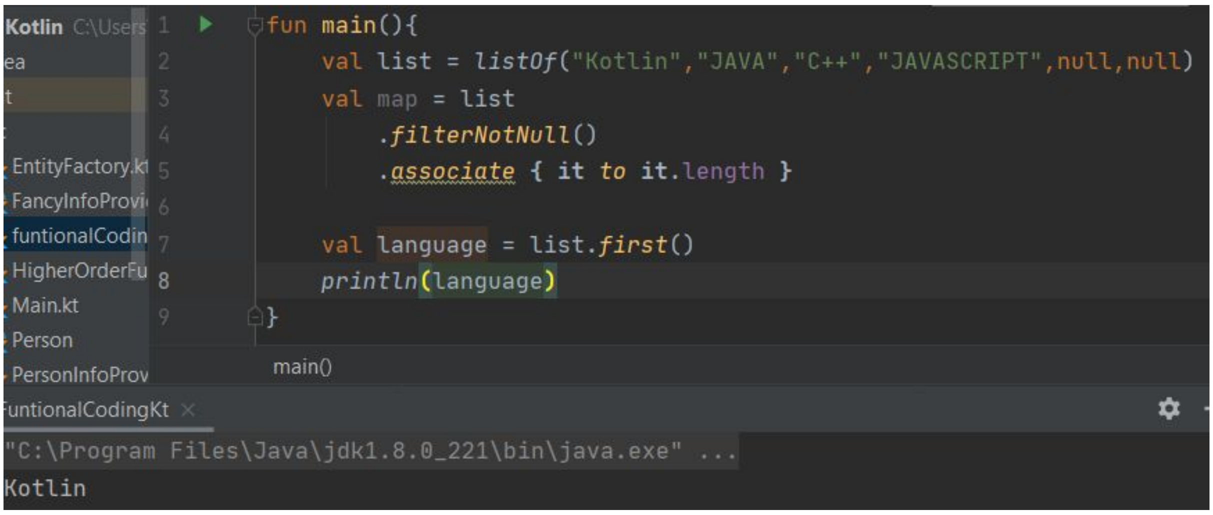
take first 3 elements



```kotlin
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    list
        .filterNotNull()
        .associate { it to it.length }
        .forEach { it: Map.Entry<String, Int>
        println("${it.value}, ${it.key}")
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
6, Kotlin
4, JAVA
3, C++
10, JAVASCRIPT
```

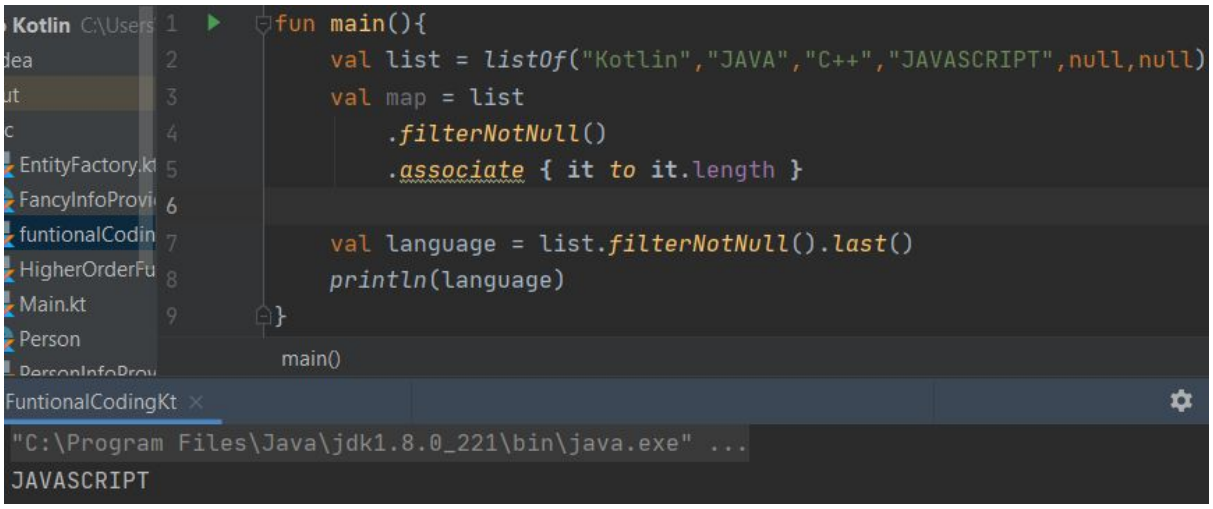
56. Pulling out required element from the collection



```kotlin
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    val map = list
        .filterNotNull()
        .associate { it to it.length }

    val language = list.first()
    println(language)
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
Kotlin
```

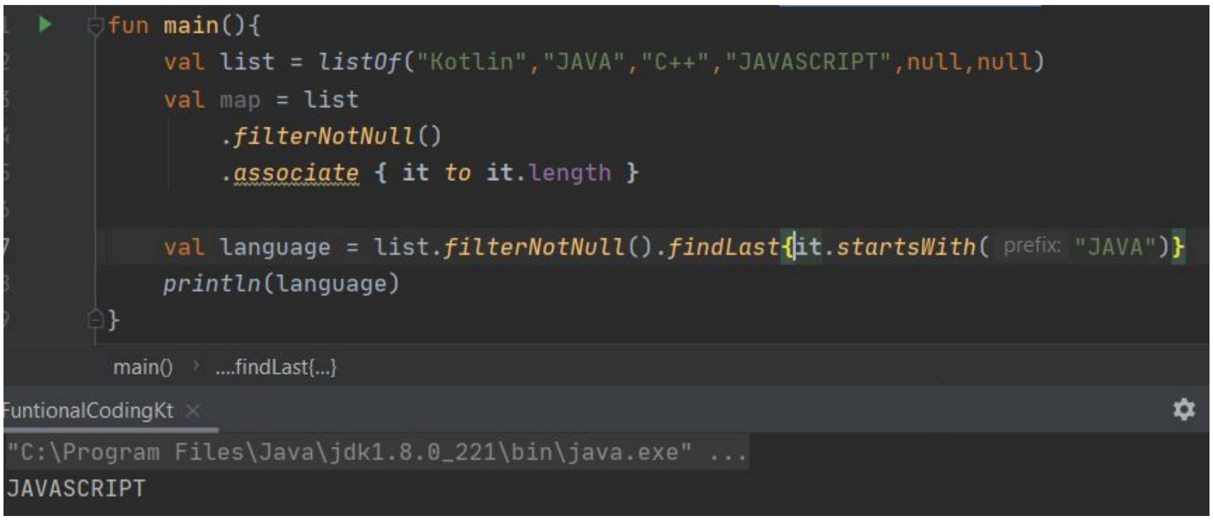
57. Last NotNull value from the List (Collection)



```
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    val map = list
        .filterNotNull()
        .associate { it to it.length }

    val language = list.filterNotNull().last()
    println(language)
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
JAVASCRIPT
```

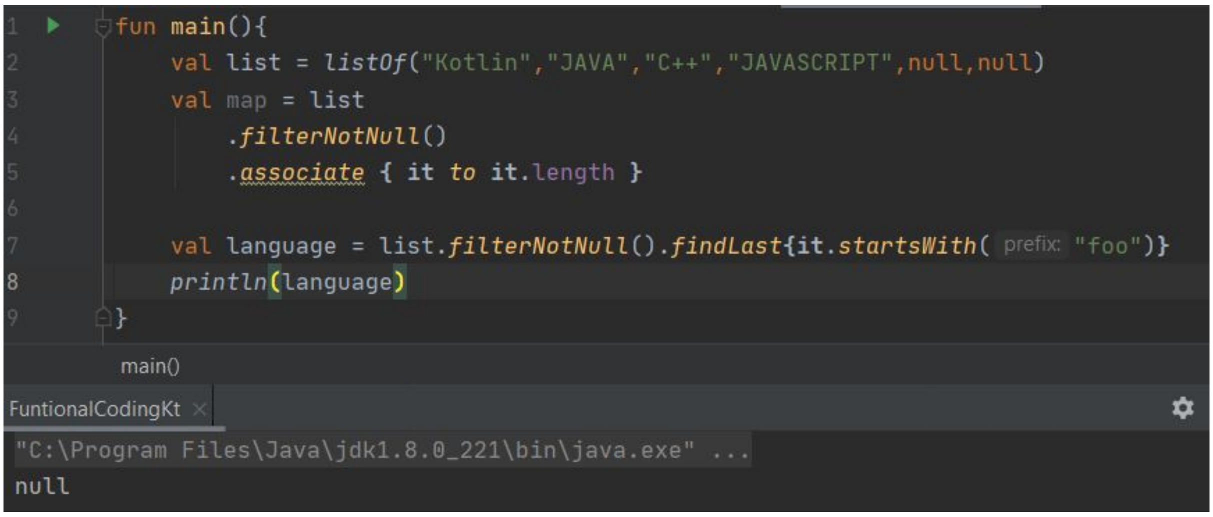
58.



```
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    val map = list
        .filterNotNull()
        .associate { it to it.length }

    val language = list.filterNotNull().findLast{it.startsWith( prefix: "JAVA")}
    println(language)
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
JAVASCRIPT
```

59.String not in Collection then what happens: Returns NULL

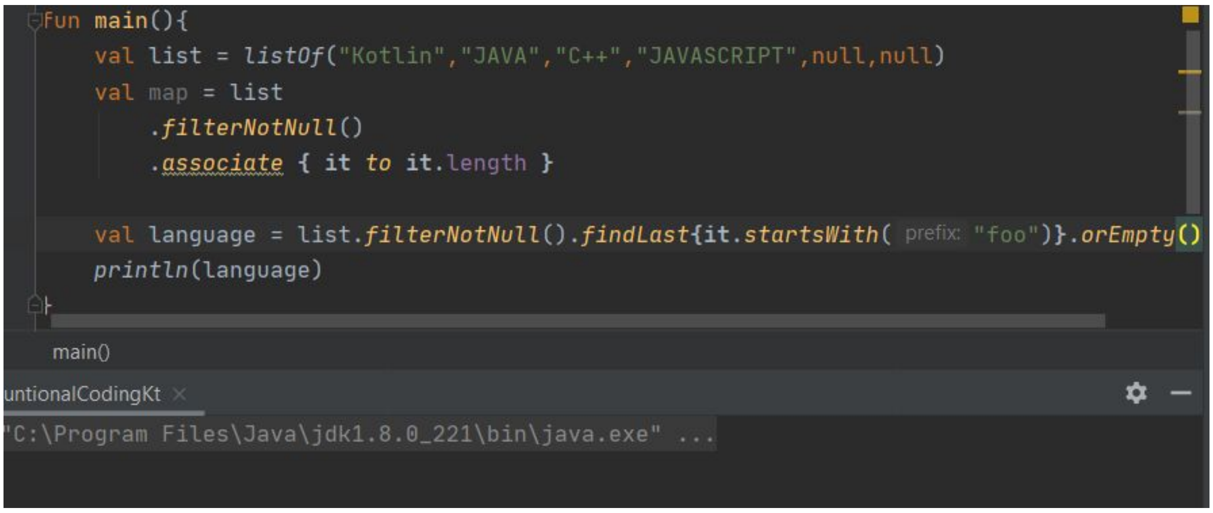


```
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    val map = list
        .filterNotNull()
        .associate { it to it.length }

    val language = list.filterNotNull().findLast{it.startsWith( prefix: "foo")}
    println(language)
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
null
```

60.What if we don't want to work with NULL

Returns Empty String : Not the NULL



```
fun main(){
    val list = listOf("Kotlin","JAVA","C++","JAVASCRIPT",null,null)
    val map = list
        .filterNotNull()
        .associate { it to it.length }

    val language = list.filterNotNull().findLast{it.startsWith( prefix: "foo")}.orEmpty()
    println(language)
}
```

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...
```