

Kotlin Journey- II

Data Classes

- Reduces boilerplate for the model classes or POJO we made in Java.
- Reduces around 70-80 lines in one liner
- Have capability of all setter, getter, hashCode(), equals() internally
- Making copies is super easy



```
//Represent POJO for Person class  
data class Person(val name: String, val age: Int, var email: String, var phone: Long)
```



```
val person1 = Person(3, "abc@gmail.com")  
  
val person1Copy = person1.copy()  
  
val person1With30 = person1.copy(age = 30)  
  
val person4WithEmail = person4.copy(email = "person4@gmail.com")
```

- Not always good Auto Value or Builder Pattern can create a mess in Java call.
- Use @JvmOverloads for having automatic generated multiple constructors.

Sealed Class

- Enum Class on steroids
- Extensions of Enums
- It can be subclass but only in same file
- Can not instantiate from outside
- Can hold data
- Standard java enum can't support different kind of associated value per case.

```
enum class BasicScreenState {  
    ERROR,  
    LOADING,  
    DATA  
}
```

To

```
sealed class ScreenState {  
    class Error : ScreenState()  
    class Loading : ScreenState()  
    data class Data(val someData: SomeData) : ScreenState()  
}
```

```
fun setScreenState(screenState: ScreenState) {  
    when(screenState) {  
        is ScreenState.Error -> { /* set error state in the view */ }  
        is ScreenState.Loading -> { /* set loading state in the view */ }  
        is ScreenState.Data -> {  
            /* hide loading or error states in the view and display data*/  
            //sometextView.text = screenState.someData.name  
        }  
    }  
}
```

Standard.kt

with, let, apply, run, also




Great Powers


Let

- Called object passed in via argument
- Return type is whatever the lambda involved in this let code will return
- The last line if individually written is considered as returned value of the lambda blocks.
- Used when we have to convert one data type to other with some operations
- Checking and avoid nullability

Use of Let Keyword




```
val answerToUniverse = strBuilder.let {  
    it.append("Douglas Adams was right after all")  
    it.append("Life, the Universe and Everything")  
    42  
}  
// using 'let' to only print when str is not null  
str?.let { print(it) }
```




```
val person = Person("Edmund", 42)  
val result = person.let { it.age * 2 }  
println(person)  
println(result)
```

Output:

Use of Let Keyword



```
val answerToUniverse = strBuilder.let {  
    it.append("Douglas Adams was right after all")  
    it.append("Life, the Universe and Everything")  
    42  
}  
// using 'let' to only print when str is not null  
str?.let { print(it) }
```



```
val person = Person("Edmund", 42)  
val result = person.let { it.age * 2 }  
println(person)  
println(result)
```


Output:

Person(name = Edmund, age = 42)

84

Apply

- Apply is used for post construction configuration
- Objects is passed this rather than argument.
- Used for initialization and configuration of objects in a easier way.




```
val parser = ParserFactory.getInstance().apply{
    setIndent(true)
    setLocale(Locale("hr", "HR"))
}

val medicine = Medicine.getInstance().apply{
    setPrice(293)
    setLocale(Locale("hr", "HR"))
}
```

Use of Apply keyword

Run

- With function literal receiver.
- Do return values
- Passed function to argument with this as receiver,



```
val person = Person("Edmund", 42)
val result = person.run { age * 2 }
println(person)
println(result)
```


```
//Output
Person(name=Edmund, age=42)
84
```

With

- Call multiple methods on a single object
- Not works with nullable
- Passing arguments



```
with(student) {  
    studentName.text = name //the property in Student called directly instead of calling student.name or  
    student.getName()  
    studentSubject.text = subject  
    Picasso.with(itemView.context).load(image).into(studentImage)  
}
```



```
fun startIntent(s : Student){
    val intent = Intent(this,HomeActivity::class.java)
    with(intent){
        putExtra(NAME,name) // No need to use intent object anymore
        putExtra(SUBJECT,subject)
        putExtra(IMAGE,image)
    }
    startActivity(intent)
}

fun receiveIntent(i : Intent){
    with(i){
        val name = getStringExtra(NAME)
        val subject = getStringExtra(SUBJECT)
        val image = getStringExtra(IMAGE)
    }
}
```

Also


- Also do this with the object.
- Can be used for debugging.



```
kittenRest
    .let { KittenEntity(it.name, it.cuteness) }
    .also { println(it.name) }
    .also { kittenCollection += it }
    .let { it.id }
```

Lateinit and lazy


- Initialization like java are no longer important
- Variables can be initialized later on in the code with full liberty
- For this purpose you have 2 keywords: **lateinit** and **lazy**
- **Lateinit:** It means you will be initializing the value of variable before accessing it. If not *kotlin.UninitializedPropertyAccessException* is there
- **Lateinit cannot be used with primitive data type.**



```
lateinit var test: String

fun doSomething() {
    test = "Some value"
    println("Length of string is "+test.length)
    test = "change value"
}
```

- Lazy keyword: It's same as lazy initialization. Your variable will not be initialized unless you use that variable in your code. It will be initialized only once after that we always use the same value.



```
val test: String by lazy {
    val testString = "some value"
    testString
}

fun doSomething() {
    println("Length of string is "+test.length)
}
```

Inheritance

- All classes implicitly inherits Any, default superclass for the present class
- Inheritance works normally:

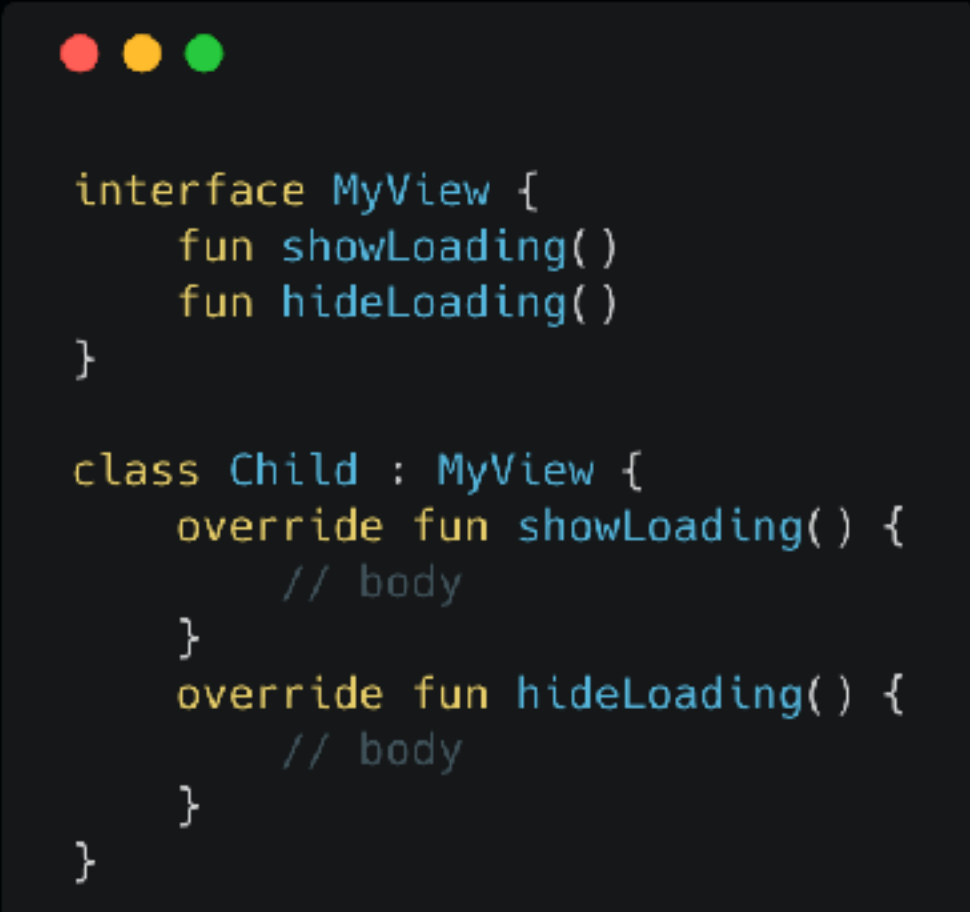


```
class MyView(context: Context?, attrs: AttributeSet?) : View(context, attrs) {  
  
    override fun onDraw(canvas: Canvas?) {  
        //Drawing occurs here  
    }  
  
}
```

- Overridden methods are pre appended with keyword of override.

Interfaces

- They can have abstract or method implementations
- Similarly for the properties or the fields that is it can also be abstract or provide implementation to accessors.
- The interfaces can inherit the other interface as well



```
interface MyView {  
    fun showLoading()  
    fun hideLoading()  
}  
  
class Child : MyView {  
    override fun showLoading() {  
        // body  
    }  
    override fun hideLoading() {  
        // body  
    }  
}
```



```
interface MyInterface {  
    val prop: Int // abstract  
  
    val propertyWithImplementation: String  
        get() = "foo"  
  
    fun foo() {  
        print(prop)  
    }  
}  
  
class Child : MyInterface {  
    override val prop: Int = 29  
}
```

Example with abstract and implementation of field in interfaces

Thank You 🙋