

Kotlin Primer

Ramp up

Introduction

First and almost, why did i choose Kotlin ?

- Because i get more than half of Effective Java implemented by it.
- It's a modern language with less overhead than java
- You write less and safer code

An overview of what u can expect

- Basic syntax (Functions, properties vs fields, collections, classes. ...)
- Std. lib functions (let, apply, run, with,....)
- Nullability
- Delegates (Observable, Vetoable, Lazy, property and class delegates)
- Sealed & data classes
- Destructuring
- Variance – the big difference
- Extensions

Basic syntax - Function Declaration

In Kotlin, in contrast to Java you specify first the **name** of the variable followed by it's **type**

```
public fun sum(a: Int, b: Int): Int {  
    return a + b  
}
```

```
fun sum(a: Int, b: Int): Int {  
    return a + b  
}
```

```
fun sum(a: Int, b: Int) = a + b
```

Visibility modifier can be omitted, if it's public

Method block can be omitted if it's a simple one-liner

Basic syntax – Functions & default values

You can finally use **default values** for parameters passed into a **function** or **constructor** and you can (must) also use **named parameters**

```
public fun sum(a: Int = 10, b: Int): Int {  
    return a + b  
}
```


```
sum(b = 2)  
sum(a = 2, b = 5)
```

Basic syntax – Function types

In Kotlin, functions are „*first-class citizen*“, which means:

- A function can be assigned to a variable
- Passed as an argument to another function
- Returned from a function

But Kotlin is also statically typed, therefore functions have a type, which is called **function type**

- 
- () -> Unit
 - (Int) -> Double
 - () -> () -> Unit

```
class MyFunction: () -> Unit{  
    override fun invoke() {  
        println("MyFunction called")  
    }  
}
```

Basic syntax – Function literals

Another way of providing a function is to use a **function literal**. A function literal is a special notation used to simplify how a function is defined.

- **Lambda expressions**
- **Anonymous functions**

Lambda expression is a short way to define a function.

```
val greetings: () -> Unit = { println("Greetings") }
```

```
val divideByHalf: (Int) -> Int = { x -> x / 2 }
```

Anonymous function is an alternative way to define a function

```
val greetings = fun() { println("Greetings") }
```

```
val divideByHalf = fun(x: Int) = x/2
```

Basic syntax – Higher Order Functions

A higher order function is a function which takes other functions as parameter or returns a functions.

```
fun isOdd(x: Int) = x % 2 != 0
```

This function can now be used as an argument in another function in different ways:

Given the following list: `val listOfNumbers = mutableListOf<Int>(1,2,3,4)`

```
listOfNumbers.filter(::isOdd)
```

```
listOfNumbers.filter { isOdd(it) }
```

```
val predicate : (Int) -> Boolean = ::isOdd  
listOfNumbers.filter(predicate)
```

```
listOfNumbers.filter { it % 2 != 0 }
```


Basic syntax – Function composition

Consider the following function:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {  
    return { x -> f(g(x)) }  
}
```



compose(f, g) = f(g(*))

Now given the following callable references:

```
fun isOdd(x: Int) = x % 2 != 0  
fun length(string: String) = string.length
```

We can compose them by:

```
val oddLength = compose(:isOdd, ::length)
```



```
listOfStrings.filter(oddLength)
```

Basic syntax – Classes

A **big difference** to Java regarding **classes** in **Kotlin** is the fact that they **are final by default**.

There are 4 types of classes in Kotlin:

- Normal class
- Data class
- Sealed class
- Enum class

In contrast to Java and very similar to Swift, Kotlin distinguishes between a **primary** and **secondary** constructor

The rule is simple: Every secondary constructor has to call the primary constructor!

Basic syntax – Classes (Primary & Secondary Constructors)

```
class Test{} or class Test
```

```
class Test constructor(variable: Any)
```

```
class Test constructor(variable: Any){  
    init{  
        val instanceVariable = variable  
    }  
}
```

```
class TestClass constructor(val variable1: Any, val variable2: Any){  
  
    constructor(variable1: Any) : this(variable1, "")  
  
    constructor() : this("", "")  
  
}
```

Basic syntax – Class inheritance

As already mentioned every class in Kotlin is **final** by default, to open up a class for inheritance we have to use the **open** keyword.

```
open class Base {  
    open fun a(){}  
    fun b(){}  
}
```



```
class Derived : Base(){  
    final override fun a() {}  
}
```

The **same rules** also apply for **overriding properties**. With one exception:

Val properties can be overridden by var variables but not the otherway around.

Basic syntax – Extending Android classes

Sometimes it is necessary to implement your own view which extends from an Android class. Depending on target API level you have to provide 3 different constructors.

```
class AwesomeAndroidView: View{  
    constructor(context: Context): super(context)  
    constructor(context: Context, attr: AttributeSet) : super(context,attr)  
    constructor(context: Context, attr: AttributeSet, defStyleAttr: Int) : super(context,attr,defStyleAttributeSet)  
    ...  
}
```



@JvmOverloads

```
class AwesomeAndroidView @JvmOverloads constructor(context: Context, attr: AttributeSet? = null,  
defStyleAttributeSet: Int = 0) : View(context,attr,defStyleAttributeSet){}
```

Basic syntax – Object keyword

In Kotlin there is **no static keyword** nor are there **anonymous classes**.

Instead Kotlin introduced a new keyword **object** which serves two purposes.

- Object declaration
 - Replacement for singleton pattern
- Object expression
 - Replacement for anonymous classes

Basic syntax – Object declaration

Objects can't have any constructors, therefore you can't pass any argument to it but they can have members as normal classes.

```
object RedditService{  
    private lateinit var api : RedditAPI  
    init {  
        api = Retrofit.Builder()  
            .baseUrl(URL)  
            .build()  
            .create(RedditAPI::class.java)  
    }  
    fun getRedditPosts(title: String, sortOrder: String = "top") = api.getRedditPosts()  
}
```

Basic syntax – Object expression

Object expression is a structure that creates a single instance of an object:

```
val clickPt = object {  
    var x = 10  
    var y = 10  
}
```

You will use it a lot throughout your code as this pattern is used to substitute the Java anonymous classes.

```
username.addTextChangedListener(object : TextWatcher {  
    //...  
    override fun afterTextChanged(editable: Editable) {  
        //...  
    }  
}))
```


Basic syntax – Companion object

Companion objects are a brother of the object declaration. It works the same, but it takes the name of the enclosing class.

```
class TestFragment: Fragment(){  
  
    companion object Factory {  
  
        fun getInstance(arg: Map<String,String>): TestFragment{  
            val fragment = TestFragment().apply {  
                val bundle = Bundle()  
                arguments = bundle  
            }  
            return fragment  
        }  
    }  
}
```

Basic syntax – Properties vs Fields

First and all, in Kotlin we distinguish between **mutable** (**var**) & **immutable** (**val**) variables.

So how to properties (fields) look in Java and Kotlin?

```
public String name = "Chris";
```



```
var name: String = "Chris"
```

```
private String name = "Chris";
```

```
public String getName() {  
    return name;  
}
```

```
public void setName(String name) {  
    this.name = name;  
}
```

They both look very similiar, do they?

Actually these are 2 completely different concepts.

Basic syntax – General structure of properties

That is how a property in Kotlin is defined:

```
[val | var ] <Property Name> : <Property Typ> = <Property Initialisation>  
    <getter>  
    <setter>
```

There is no need to define custom getter or setter, you can directly control the visibility or behaviour of a property while u declaring it.

```
var name: String = "Chris"  
private set
```

```
var name: String = ""  
    get() = "Chris"  
private set
```

Basic syntax – Properties and fields

Normally there are no **(backing) fields** on class level in Kotlin, only properties. But you have the possibility to define a backing field when you are working inside a **customer getter** or **setter**.

This will then look like as follows:

```
var name: String = ""  
get() = field  
set(value){  
    if (value != null){  
        field = value  
    }  
}
```

We will get back to the properties syntax when we are looking into **Delegates**

Basic syntax – Collections

- **List:**
 - `listOf(T)` or `mutableListOf(T)`
- **Set**
 - `setOf(T)` or `mutableSetOf(T)`
- **Map**
 - `mapOf(key 'to' value, key 'to' value,...)` or `mutableMapOf(..)`

You can access elements inside collections by the use of the property syntax similar to JavaScript.

```
var mutableList = mutableListOf("1","2","3")  
println(mutableList[1])
```

End of section: Basic syntax
Any questions ?

Std. lib. Functions – or how one-liners can change the world

Before we dive into the programming concepts of kotlin lets look into some special functions provided by Kotlin which will help us in different ways.

The **standart.kt** library provide some more or less useful (extension) functions, most of these functions are just one-liners but with a great effect.

A month ago, Google released an alpha version of Android- kotlin-extensions (Android KTX), which provides some usefull extension function purely for Android development. We will not cover them here as they are still in version 0.2

<https://github.com/android/android-ktx/>

Std. lib. Functions – Let

```
fun <T, R> T.let(f: (T) -> R): R = f(this)
```

```
DbConnection.getConnection().let { connection ->
}
// connection is no longer visible here
```

You in combination with the null-check operator:

```
val map : Map<String, String> = ...
val config = map[key]
config?.let {
    // This block will not be executed if „config“ is null
}
```


Std. lib. Functions – Apply

```
fun <T> T.apply(f: T.() -> Unit): T { f(); return this }
```

With apply() we can substitute the “Builder” pattern or simply make our code more readable.

```
val recyclerView: RecyclerView = RecyclerView().apply{  
    setHasFixedSize = true  
    layoutManager = LinearLayoutManager(context)  
    adapter = MyAdapter(context)  
    clearOnScrollListener()  
}
```

Std. lib. Functions – Run

```
fun <T, R> T.run(f: T.() -> R): R = f()
```

Run() should only be used with lambdas which do not return any values , but only generate sideeffects.

```
webview.settings?.run {  
    javascriptEnabled = true  
    databaseEnabled = true  
}
```

Std. lib. Functions – Also

```
fun <T> T.also(block: (T) -> Unit): T
```

With **also** you say „also do this with the object“.

Also() passes the object as parameter and returns the same object (not the result of the lambda).

```
val person = Person().also {  
    it.name = "Tony Stark"  
    it.age = 42  
}
```

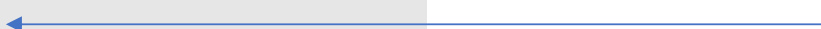
Std. lib. Functions – With

```
fun <T, R> with(receiver: T, f: T.() -> R): R = receiver.f()
```

It just helps eliminating the repetitive code for setting properties.

By default with() return the result of the last line, if you want to return the object you need to add **this** as the last line.

```
val person = with(Person()){  
    this.age = 42  
    this.name = "Tony Stark"  
    this  
}
```



*Note **with()** doesn't work with nullable variables*

Std. lib. Functions – Use

```
fun <T : Closeable, R> T.use(block: (T) -> R): R
```

Use function is the equivalent of Java's try-with-resources. It applies to all types of closable instances. It automatically closes the resource (receiver) on exit.

Java style:

```
try(FileReader reader = new FileReader("Input.txt")){  
    // Read file  
}catch (IOException e){  
}  
//automatically closed
```

Kotlin style:

```
FileReader("Input.txt").use {  
    // Read File  
}  
  
// Automatically closed
```

Std. lib. Functions – takelf

```
fun <T> T.takelf(predicate: (T) -> Boolean): T? = if (predicate(this)) this else null
```

Takelf is a filter for a single value, in combination with the Elvis Operator (?:) you can handle the else case

```
val name: String = "Chris"  
val index = name.indexOf("C").takelf { it > 0 } ?: 0
```

Std. lib. Functions – takeUnless

```
fun <T> T.takeUnless(predicate: (T) -> Boolean): T? = if (!predicate(this)) this else null
```

TakeUnless is the exact opposite to takeIf(). It takes an inverted predicate.

```
val name: String = "Chris"  
val index = name.indexOf("C").takeUnless { it < 0 } ?: 0
```

Std. lib. Functions – When to use what

- **Also** : Additional processing on an object in a call chain
- **Apply** : Post-construction configuration
- **Let** : conversion of value (null check)
- **Run** : Execute lambda with side-effects and no result
- **With** : Configure object created somewhere else

Be careful when using these functions to avoid potential problems:

- Do not use **with** on nullable variables.
- Avoid nesting **apply**, **run** and **with** as you will not know what is the current **this**.
- For nested **apply** and **let**, use named parameters instead of **it** for the same reason
- Avoid **it** in long call chains as it is not clear what it represents.

Cheat Sheet - Introduction

```
class MyClass {  
  fun test() {  
    val str: String = "..."  
  
    val result = str.let {  
      print(this)  
      print(it)  
      42  
    }  
  }  
}
```

This (Receiver):

This is an instance of MyClass (this@MyClass) while test() a method of MyClass is.

It (Argument):

It is the String “...” on which we executed **let**.

42 (Return Value):

42 is the result which will be returned from the block

Function	Receiver (this)	Argument (It)	Result
Let	This@MyClass	String(“...”)	Int(42)

Cheat Sheet – for the Std.kt

Funktion	Receiver (this)	Argument (It)	Result
Let	This@MyClass	String("...")	Int(42)
run	String("...")	n/a	Int(42)
run*	this@MyClass	n/a	Int(42)
with*	String(„...“)	n/a	Int(42)
apply	String(„...“)	n/a	String(„...“)
also	this@MyClass	String(„...“)	String(„...“)

** = No extension function. These methods have to be called in the old way*

End of section: Std. library
Any questions ?

Nullability

In Kotlin we differentiate between references that can be null and those that can't be. This concept leads to some changes in how you write your code.

```
var a = 1  
a = null
```

// Compile time error

```
var a : Int? = 1  
a = null
```

// No compile error

*Be aware that **Int** is a **class** whereas **Int?** is a **type**.*

We have different options in how we can deal with nullable values, which are shown on the following slides.

Nullability

Variant 1: Check explicitly for null (The java way)

```
val a : Int? = 2
```

```
val c = if(a != null) a else -1
```

```
val d = a ?: -1
```

Variant 2: Safe Call operator

```
val a : String? = "Hello"
```

```
val b = a.length
```

// Compile time error

```
val a : String? = "Hello"
```

```
val b = a?.length
```

Nullability

Variant 3: Safe-call operator in combination with **let**

```
val list = listOf("1","2","3","4")
for(number in list){
    number?.let {
        println(number)
    }
}
```

Nullability

Variant 4: For all NPE Lovers, the !! operator

```
var b : String? = null  
println("${b!!}.length")
```

// This will throw an NPE

```
b?.let {  
    b!!  
}
```

Variant 5: Do it yourself

```
fun <T> T?.or(default: T): T = if (this == null) default else this
```

```
var variable: String? = "Hello"  
println(variable?.length.or(-1))
```

Nullability & type checks

The nullability also applies for type checks with **as** & **is**

is = instanceOf()

as = smart casting

```
var obj : Int = 1

if(obj is String){
    println(obj.length)
}
```

unsafe cast

```
val x: String = y as String
```

safe cast

```
val x: String? = y as String?
```

safe cast

```
val x: String? = y as? String
```


End of section: Nullability
Any questions ?

Delegates

«It is said that inheritance is useful only in case of having Donald Trump as a father»

The problem with wrongly designed or wrongly documented inheritance is that:

«The interaction of inherited classes with their parents can be surprising and unpredicable if the ancestor wasn't designed to be inherited from»

Delegates – Standard delegation

```
private val textView : TextView by lazy {  
    val view = findViewById(R.id.tvName) as TextView  
    view  
}
```

```
var name: ViewState by Delegates.observable(ViewState()) { property: KProperty<*>, old, new ->  
    updateView(new)  
}
```

```
var name: ViewState by Delegates.vetoable(ViewState()) { property: KProperty<*>, old, new ->  
    new != null  
}
```

```
var name: ViewState by notNull()
```

Delegates – Property delegation

The general structure of delegated properties looks like this:

```
[val | var] <Property Name> : <Property Typ> by <Expression>
```



The term **Expression** after **by** presents the **delegate**. Calls to `get()` on the property will be forwarded to the delegate. Therefore the delegate has to provide a **`get()`** method **for immutable properties** and a **`set()`** method **for mutable properties**

```
public interface ReadOnlyProperty<in R, out T> {  
    public operator fun getValue(thisRef: R, property: KProperty<*>): T  
}
```

```
public interface ReadWriteProperty<in R, T> {  
    public operator fun getValue(thisRef: R, property: KProperty<*>): T  
    public operator fun setValue(thisRef: R, property: KProperty<*>, value: T)  
}
```

Delegates – A useful example

Problem:

Every mobile developer knows that it is really important to save the state of objects or properties in fragments or activities, since both will not live the whole time.

Solution:

Write a delegate that will automatically save and restore the state of properties and persistence them in the appropriate bundle

Excercise:

Checkout branch ***chapter_02_section_01_property_delegates_exercise*** and search for „*chapter_02_section_01_property_delegates_exercise*» and implement the missing logic

Delegates – Class delegation

You can delegate **interface** methods of a class to another class. It's like inheritance, but with 2 major differences.

- You can delegate multiple classes
- You only share the interface methods, no other methods and variables

Delegates – Class delegation

The Java way:

```
interface Base1{  
    fun get1(): String  
}
```

```
interface Base2{  
    fun get2(): String  
}
```

```
class DerivedByInheritance(): Base1,Base2{  
    override fun get1(): String = "get1"  
    override fun get2(): String = "get2"  
}
```

The Kotlin way:

```
class Base1Impl: Base1{  
    override fun get1(): String = "get1"  
}
```

```
class Base2Impl: Base2{  
    override fun get2(): String = "get2"  
}
```



```
class DerivedByDelegates(private val base1: Base1, private val base2: Base2): Base1 by base1, Base2 by base2
```

Delegates – Map Delegate

```
class MyMessagingService : FirebaseMessagingService() {  
  
    override fun onMessageReceived(message: RemoteMessage?) {  
        super.onMessageReceived(message)  
        val data = (message?.data ?: emptyMap()).withDefault { "" }  
        val title = data["title"]  
        val content = data["content"]  
        print("$title $content")  
    }  
}
```


Delegates – Map Delegate

```
class NotificationParams(val map: Map<String, String>) {  
    val title: String by map  
    val content: String by map  
}
```

```
override fun onMessageReceived(message: RemoteMessage?) {  
    super.onMessageReceived(message)  
    val data = (message?.data ?: emptyMap()).withDefault { "" }  
    val params = NotificationParams(data)  
    print("${params.title} ${params.content}")  
}
```

Delegates – Exercise 02 (Class delegation)

Problem:

Currently our LoginViewModel depends on the PreferenceHolder, which is an abstraction of the SharedPreferences (simple android persistence layer). The abstraction is implemented by explicit delegation.

Solution:

Rewrite the PreferencesHolder to use the build-in delegation functionality of Kotlin

Excercise:

Checkout branch ***chapter_02_section_02_class_delegates_exercise*** and search for „*chapter_02_section_02_class_delegates_exercise*» and implement the missing logic

End of section: Delegates
Any questions ?

Sealed & Data classes – Data classes

Data classes are basically Java Pojo classes. In case where you implement a class which only **holds data**, use the Kotlin's **data class**.

When you define a **data class**, Kotlin will automatically generate the following methods for you

- getter & setter
- hashCode & equals
- toString
- copy
- componentX

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

Sealed & Data classes – Data classes

Java

```
public class User {  
    private String name;  
    private int age;  
  
    public User(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
  
        User user = (User) o;  
  
        if (age != user.age) return false;  
        return name.equals(user.name);  
    }  
    @Override  
    public int hashCode() {  
        int result = name.hashCode();  
        result = 31 * result + age;  
        return result;  
    }  
    @Override  
    public String toString() {  
        return "User{" +  
            "name=" + name + " +  
            ", age=" + age +  
            '}'  
        }  
    }  
}
```

Kotlin

==

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

You spot the difference ?

Sealed & Data classes – Data classes

Class destructuring

Destructuring is the concept of treating objects as a set of separate variables.

```
val (name, age) = user  
val (_, age) = user
```



```
val name = person.component1()  
val age = person.component2()
```



```
operator fun Person.component1() = getName()  
operator fun Person.component2() = getAge()
```

```
val users = listOf<User>()  
for((name, age) in users){  
    ...  
}
```

Be aware that the reference is done by positional data

Class copy

```
val user = User("Chris", 36)  
val copy = user.copy(age = 21)
```

Sealed & Data classes – Sealed classes

Before we look into Sealed classes lets compare the enums of Java & Kotlin

Java's enum

```
public enum ViewState {  
    ERROR,  
    LAODING,  
    DATA  
}
```

Kotlin's enum

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

Both look very similar. Enums are already pretty concise in Java, so Kotlin doesn't save us any boilerplate code. Regarding functionality, they are also very simple as there is not much you can with them...

```
fun handleViewState(viewState: ViewState): Unit = when(viewState){  
    ViewState.ERROR -> { /* set error state in the view */ }  
    ViewState.DATA -> { /* hide loading state and process data ? */ }  
    ViewState.LOADING -> { /* show loading indicator */ }  
}
```

Sealed & Data classes – Sealed classes

So in essence, if you want to use enums for something simple that doesn't need to hold any data or different types of data per value, then they are fine.

But if we need something more specific we need something else.

And this is where Kotlin's **Sealed classes** come into play. (*Sealed classes are comparable to Swift's enum*)

- *Sealed classes are used for representing restricted class hierarchies.*
- *They are, in a sense, an extension of enum classes.*
- *A subclass of a sealed class can have multiple instances which can contain state.*

Sealed & Data classes – Sealed classes

*In short, **Sealed classes** are good when returning different but related results.*

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

```
fun handleViewState(viewState: ViewState): Unit = when(viewState){  
    is ViewState.Error -> { /* show error screen */ }  
    is ViewState.Loading -> { /* show loading indicator */ }  
    is ViewState.Data -> { handleData(viewState.someData) }  
}
```

Sealed Classes - Exercise

Currently our LoginViewModel depends on the class PreferencesHolder in order to persist the valid username and password. Although we abstracted the Android Implementation „SharedPreferences“ by our PreferencesHolder we could have also modeled the whole persistence directly in the Fragment, without relying on the LoginVieModel.

Excercise:

*Checkout branch **chapter_02_section_03_sealed_classes_exercise** and search for „chapter_02_section_03_sealed_classes_exercise» and replace all occurences of the LoginViewModel.ViewState enums by implementation which is based on Sealed Classes*

End of section: Sealed Classes
Any questions ?

Variance – What is it?

- Most of the modern programming languages support **subtyping** which allows us to implement hierarchies like **a Cat Is-An creature**.
- In **Java** we use **extend** to *exchange/expand* the behaviour of an **existing class** or use **implements** to *provide* an implementation for an **interface**.
- The word **variance** in this context is used to describe how **subtyping in** complex aspects like:
 - **Method return type**
 - **Type declaration**
 - **Arrays relates to the direction of inheritance**of the involved classes.

Variance – Covariance in Java

```
public class Creature {}
```

```
public class Insect extends Creature {}
```

In **Java** a **overriding method** needs to be **covariant** in it's return type

```
abstract public class Veterinary {  
    abstract public Creature treat();  
}
```

```
public class InsectVeterinary extends Veterinary {  
    @Override  
    public Insect treat() {  
        return null;  
    }  
}
```

Subclasses can be **cast up the inheritance tree**, while **downcasting** will cause an error

```
(Insect) new Creature()
```

// Error

```
(Creature) new Insect()
```

Variance – Covariance in Java

In **Java arrays** are **covariant**.

But, there is a big problem with this:

```
Integer[] numbers = {1,2,3,4};  
Object[] objects = numbers;  
objects[0] = "String";
```

// Runtime exception: Exception in thread "main"...

Variance – Generic Collections

- As of **Java 1.5** we can use **Generics** in order to tell the compiler which elements are supposed to be stored in our collections. Unlike Arrays, **generic collections** are **invariant** in their parameterized type **by default**.
- This means you can't substitute `List<Creature>` with `List<Insect>`, it won't even compile.
- Fortunately, the *user can specify the variance of type parameters himself when using generics*, which is called **use-site-variance**.

Variance – Covariant collections In Java

The following code snippet shows how to declare a **covariant** list of Creatures and assign a list of insect to it.

```
List<Insect> insects = new ArrayList<>();  
List<? extends Creature> creatures = insects;
```

Such a **covariant list** still differs from a an array, because the covariance is encoded in it's type parameter, which means we can **only** read from it.

```
creatures.add(new Insect());           // Won't compile
```


Variance – Contravariant collections In Java

The following code snippet shows how to declare a contravariant list of creatures.

```
List<Creature> creatures = new ArrayList<>();  
List<? super Creature> contraVariantCreatures = creatures;
```

Like with covariant lists, we don't know for sure which type the list contains. The difference is, **we can't read from** it, since it is unclear if we'll get a creature or just a plain object. But know we **can write to it**, as we know that at least a creature may be added.

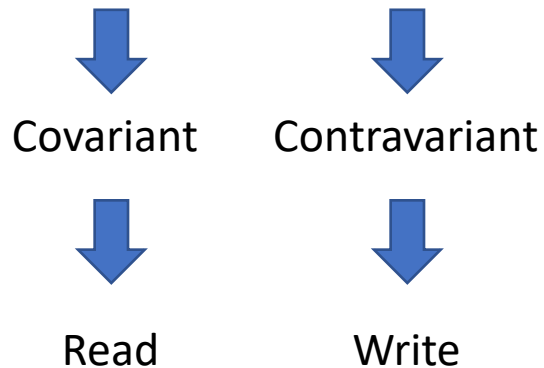
```
contraVariantCreatures.add(new Insect());
```

```
Creature creature = contraVariantCreatures.get(0); // Won't compile
```

Variance – Summary of Java Variance

Joshua Bloch created a rule of thumb in his book **Effective Java**:

*“Producer – extends, consumer – super (**PECS**)”*



Variance – of type collections in Kotlin

Kotlin is different from Java regarding generics and also arrays.

- The easiest difference is that **arrays** in **Kotlin** are **invariant**

```
var stringArray: Array<String> = arrayOf()
```

```
var objectArray: Array<Object> = stringArray // Won't compile
```

But, is there a way to work safely with subtyped arrays ?

Variance – Type collections in Kotlin

As you have seen, Java uses the „wildcard types“ to make generics variant.

Which is called ***use-site-variance***

In Kotlin we use ***declaration-site-variance***

The generic parameter T in Kotlin can be marked as „only produced“ with the **out** keyword, which makes T **covariant** or can be marked as „only- consumed“ with the **in** keyword, which makes T **contravariant**.

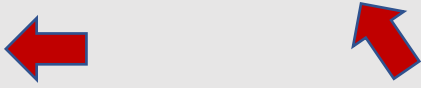
“Consumer in, producer out (CIPO)”

Variance – Covariance in Kotlin

The problem with covariance is with the mutability after upcasting. Covariant type parameters - not only setters, but on any **in** position (*public method parameters or public properties*) are a potential source of errors.

This is why Kotlin prohibits covariant type parameters used on **in** positions.

```
class Container<out T> constructor(var element: T){  
    fun set(new: T) {  
        element = new  
    }  
    fun get(): T = element  
}
```


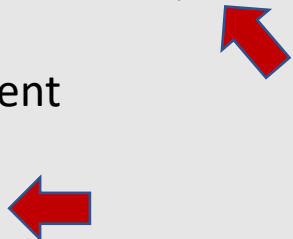


```
class Container<out T> constructor(private var element: T){  
  
    fun get(): T = element  
}
```

Variance – Contravariance in Kotlin

You might guess that contravariant parameters, which are made using the **in** keyword are only allowed on **out** positions.

```
class Container<in T> constructor(var element: T){  
    fun set(element: T){  
        this.element = element  
    }  
    fun get(): T = element  
}
```



```
class Container<in T> constructor(private var element: T){  
  
    fun set(element: T){  
        this.element = element  
    }  
}
```

Variance – Type projections in Kotlin

Sadly it is not always enough to have the opportunity of declaring a type parameter **T** as either **in** or **out**.

As an alternative Kotlin also allows sort of „use-site-variance“ which is called **type-projection**.

```
class Container<T> constructor(private var element: T){  
  
    fun copy(from: Array<out T>, to: Array<T>){  
        // ...  
    }  
  
    fun fill(dest: Array<in T>, value: T){  
        // ...  
    }  
}
```

End of section: Variance
Any questions ?

Extensions – Introduction

- Similar to Swift, C# and Gosu extensions in Kotlin expand the functionality of standard classes.
- **Kotlin** supports **extensions functions** as well as **extension properties**.

Extensions – Introduction

How to we declare extension functions ?

Just write a function as you would normally, and put the name of the class (receiver) before the function name seperated by a “.”

```
fun View.visible() {  
    this.visibility = View.VISIBLE  
}
```

Receiver class (View)



Extensions – Rules – Members always win

Member functions always win.

```
class C {  
    fun foo() {  
        println("member")  
    }  
}
```

```
fun C.foo() { println("extension") }
```

It will print „member“ always



Extensions – Rules – Nullable Extensions

It is possible to define extension functions for a nullable types

```
fun Any?.toString(): String {  
    if (this == null) return "null"  
    return toString()  
}
```

Extensions – Rules – Static Extensions

You can define extension function on class level instead of instance level.

```
class Foo{  
  
    companion object  
  
    fun sayHello() = "Hello"  
  
}
```

```
fun Foo.Companion.sayBye() = "Bye"
```

Extensions – Properties

As mentioned in the beginning, you can also define extension properties for classes. The only restriction is that you can't initialize the property directly. You have to explicitly define the getter and setter for it.

```
val Foo.bar = 1
```

// error: initializers are not allowed for extension properties

```
val Foo.bar: Int  
  get() = 1
```

Extensions – Dispatching

```
interface Loggable {
```

```
    val Any.LOGGER: Logger
```

```
    get() = Logger.getLogger(javaClass.name)
```

```
    fun Any.logI(message: String){  
        LOGGER.log(Level.INFO,message)  
    }
```

```
    fun Any.logE(message: String, error: Throwable){  
        LOGGER.log(Level.SEVERE,message,error)  
    }
```

```
    //...
```

```
}
```

Dispatch Receiver

Extension Receiver

Extensions – Reified and Inline

```
inline fun <reified T : Activity> Activity.navigateTo(intentParameters: Map<String, Serializable>) {  
    val intent = Intent(this, T::class.java)  
    intentParameters.forEach { s: String, serializable: Serializable -> intent.putExtra(s, serializable)}  
    startActivity(intent)  
}
```

```
navigateTo<SessionActivity>(mutableMapOf())
```


Extension- Exercise 01

There are 2 Util classes in our application. One is the DateUtil class which provides a method to print a given „long“ value in a readable form and a ActivityUtil class which provides 2 methods to simplify the navigation from Activity to fragments.

- 1.) Move the functionality of the DateUtil class in a proper extension class on the correct receiver
- 2.) Move the functionality of the ActivityUtil class in a dispatching extension so that only Activities which implements this dispatching interface can take advantage of the extension methods.

(Note: U need to refactor the NavController to be of a generic type, so you can make use of the interface constraints)

Exercise:

Checkout branch ***chapter_02_section_05_extension_exercise*** and search for „chapter_02_section_05_extension_exercise»

Extension- Exercise 02

Until now the RedditOverviewAdapter has to call certain notify methods when the list of items gets updated. Google released a utility class called DiffUtil a year ago which removes the need to manually call these notify methods.

You can find a small tutorial about the DiffUtil class in the following link:

<https://medium.com/@iammert/using-diffutil-in-android-recyclerview-bdca8e4fbb00>

<https://developer.android.com/reference/android/support/v7/util/DiffUtil.html>

This exercise is not about the DiffUtil itself, but rather how we can combine different concepts of Kotlin to simplify and minimize repeating calls.

To correctly solve this exercise you should make use of the following aspects:

- Delegate(s)
- Dispatching Extensions
- Lambda / Higher-Order functions

Note: This exercise is a little bit tricky. (You have to implement one new class and all other modifications should only happen inside the RedditOverviewAdapter)

Checkout branch **chapter_02_section_06_advanced_extension_exercise**