

Kotlin Basics

Download IntelliJ IDEA from here (if the link is invalid just type "IntelliJ download" in google):

<https://www.jetbrains.com/idea/download/>

(Choose Community)

We will use Android Studio Kotlin REPL (Read Evaluate Print-loop) [tools -> Kotlin -> REPL]

Problem with opening REPL Run > Edit Configurations... > Templates > Java Scratch > Shorten command line to @argfile (Java 9+) and restart Android Studio.

```

.../platforms/android-studio/idea-2020.2.1/bin/idea.exe --argfile @argfile --classpath %classpath%
`CommandLineWrapper` is ill-suited for launching apps on Java 9+.
If the run configuration uses "classpath file", please change it to "@argfile".
Otherwise, please contact support.

```

var - variables

val - finals

val always preferred

No need for semicolon

Kotlin is type inferred and type safety

Meaning this won't work:

```
var x = 7
```

```
x = 7.4
```

Kotlin has no primitives only objects (Int, Float, String)

consts are compile time constants. Meaning that their value has to be assigned during compile time, unlike vals, where it can be done at runtime. This means, that consts can never be assigned to a function or any class constructor, and only to a String or primitive.

```
const val WEBSITE_NAME = "Baeldung"
```

the Kotlin compiler inlines the const val values into the locations where they're used

If you try to type it in the REPL you will get the following error: "const 'val' are only allowed on top level or in objects", like this for example:

```
const val VALUE: String = "constant"
```

```
fun main() {
    println("$VALUE is inlined")
}
```

At first glance, we might think that the Kotlin compiler gets a static field value from a class and then concatenates it with the " is inlined" text. However, since const vals are inlined, the compiler will copy the "constant" literal wherever the VALUE constant is used. This is why they must get a value in compile time. **This constant inlining is much more efficient than getting a static value from a class.**

Kotlin Nullability

In an effort to rid the world of NullPointerException, regular variable types in Kotlin don't allow the assignment of null. If you need a variable that can be null, declare it as nullable by adding ? at the end of its type - String?, Int?.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). Regular objects cannot be null (String, Int...) - prevent null pointer exception

The main advantage is that if we check a property of null objects we get a compile time and not runtime error

To bypass the compile time check use !! - but be careful from KotlinNullPointerException

If we are not sure we use - safe call - use ?. - str?.length - > will return null and not crash

When inferring types, the compiler assumes non-null for variables that are initialized with a value.

```
var inferredNonNull = "The compiler assumes non-null"
```

If we assign null on initialization the ? Type will be auto inferred

class **Nothing** - Nothing has no instances. You can use Nothing to represent "a value that never exists": for example, if a function has the return type of Nothing, it means that it never returns (always throws an exception).

There are a few techniques for using or accessing the nullable variable. One of them is safe call ?. and another one is null check !! but before figuring out the difference among both, let's understand what they are in detail first:

a: String?	a.length	a?.length	a!!.length
"cat"	Compile time error	3	3
null	Compile time error	null	NullPointerException

?. - safe call

The best way to access nullable property is safe call operator ?.

This calls the method if the property is not null or returns null if that property is null without throwing an NPE (null pointer exception).

Safe calls are useful in chains. For example, if Bob, an Employee, may be assigned to a Department (or not), that in turn may have another Employee as a department head, then to obtain the name of Bob's department head (if any), we write the following

```
bob?.department?.head?.name
```

Such a chain returns null if any of the properties in it is null.

Since the return value is null we can combine this operator with **let** function we will see later on

The !! Operator

This operator is used to explicitly tell the compiler that the property is not null and if it's null, please throw a null pointer exception (NPE). If you are sure that the property value is not null use ?. instead of !!. Usually we use this when passing a must non null value to a function. But again be careful while using it.

Elvis Operator (?:)

This one is similar to safe calls except the fact that **it can return a non-null value if the calling property is null**

The Elvis operator will evaluate the left expression and will return it if it's not null or else will evaluate the right side expression. Please note that the right side expression will only be evaluated if the left side expression is null.

Elvis is usually used to give default values in case of null:

```
val str : String? = null
val strLength = str?.length ?: -1 - > strLength equals to -1
```

We can always perform an explicit null check `if(x != null) x.do()`

In this case if x is val or local property he will be smart casted to non-nullable and all is safe. But if it is a var class property the compiler won't smart cast him and if you do it by force you will get the following error: "smart cast to '[non null type]' is impossible, because '[variable name]' is a mutable property that could have been changed by this time" - try to always use vals

If and when

if syntax like Java but each block can return it's last line - if is not a statement but an expression!

Branches of if can be blocks. In this case, the last expression is the value of a block

'if' must have both main and 'else' branches if used as an expression

println - prints on screen

Unit - Kotlin's void

```
val age = 17
val result = if(age < 16) {
    println("Too young to drive")
    "Young"
} else if(age > 16 && age < 80) {
    println("You can drive")
    "Ok"
} else {
    println("Too old to drive")
    "Old"
}
println(result)
```

In Kotlin, if is an expression: it returns a value. Therefore, there is no ternary operator (condition ? then : else) because ordinary if works fine in this role:

```
val max = if (a > b) a else b
```

when expression

Similar to switch but No need for case just write the value and -> if it is more than one line use blocks

Use **in** or **!in** for ranges - Kotlin lets you easily create ranges of values using the `rangeTo()` function from the `kotlin.ranges` package and its operator form ..

Usually, `rangeTo()` is complemented by `in` or `!in` functions.

<https://kotlinlang.org/docs/operator-overloading.html#equality-and-inequality-operators>

Binary operations

Arithmetic operators

Expression	Translated to
<code>a + b</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b)</code>
<code>a..b</code>	<code>a.rangeTo(b)</code>

Expression	Translated to
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

No need for **break** - won't got into the next case even without break

Instead of **default** use **else**

You can even call functions or operators in the cases

Same as if it can return values

```
val age = 5
val result = when(age) {
    0 -> "just born"
    in 1..3 -> "infant"
    !in 4..10 -> "Grown up"
    2 + 3 -> "five years old"
    else -> "bigger"
}
println(result)
five years old
```

You can use when without the variable name
 And not even on the same variable! It is just a collection of boolean conditions
 You can use {} inside when for multiple code lines

```
val age = 18
when {
    age <= 17 -> println("Too young")
    age >= 18 && age <= 60 -> println("the right age")
    else -> println("The golden years")
}
the right age
```

```
val age = 70
when {
    20 > 15 -> {
        print("Yoo")
        print("Yoo")
    }
    age <= 17 -> println("Too young")
    age >= 18 && age <= 60 -> println("the right age")
    else -> println("The golden years")
}
YooYoo
```

The advantage is that if one is true then the rest won't be checked - think when you want to start the app in condition that all permission granted and all

features enabled and so on, with when you can check all of them in a single block of code

```
private fun invokeLocationAction() {
    when {
        !isGPSEnabled -> latLong.text = "Please enable your gps"

        isPermissionsGranted() -> startLocationUpdate()

        shouldShowRequestPermissionRationale() -> latLong.text = "App requires location permission"

        else -> ActivityCompat.requestPermissions(
            activity: this,
            arrayOf(Manifest.permission.ACCESS_FINE_LOCATION, Manifest.permission.ACCESS_COARSE_LOCATION),
            LOCATION_REQUEST
        )
    }
}
```

Arrays

val array = **arrOf**(...) -> array of objects Kotlin uses Array<Int>

val array: Array<Long> = arrayOf(1,2,3)

joinToString() -> returns string representation of the array (toString just return the instance address).

val result = arrayOf(1,4,6,8,9)

result.joinToString()

res54: kotlin.String = 1, 4, 6, 8, 9

array[0]

Kotlin provides a selection of classes that become primitive arrays when compiled down to JVM byte-code.

You can create an array of java primitives with intArrayOf, charArrayOf....

You can sometimes find them in the Android API:

```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<String>, grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
}
```

Kotlin.collections

List\MutableList(the regular List of java)

val list = **listOf**(...)

val mutList = mutableListOf(6,8,7)

mutList.add(7)

list[0] = 90 -> error when its not a mutable list

Please note the Kotlin's list not only can't change it's size but also it's content

val set = **setOf**(6,6) only one will stay

val map = **mapOf**(Pair(1,"Moshe))

To add to the collections use the Mutable(mutable list, set, map...)

map's put function returns the overridden value

Instead of Pair your can use **to** infix function — more on infix very soon - but in short there is **to** infix extension function to most of the classes that accept another argument and return a pair with both of them and replaces the dot notation

For retrieving a value from a map, you must provide its key as an argument of the `get()` function. The shorthand `[key]` syntax is also supported. If the given key is not found, it returns null. There is also the function `getValue()` throws an exception if the key is not found in the map - if we must pass a non null reference. Additionally, you have two more options to handle the key absence:

- `getOrDefault()` returns the specified default value if the key is not found.
- `getOrElse()` the values for non-existent keys are returned from the given lambda function (more on lambda in the next chapter).

```
val numbersMap = mapOf("one" to 1, "two" to 2, "three" to 3)
println(numbersMap.get("one"))
println(numbersMap["one"])
println(numbersMap.getOrDefault("four", 10))
println(numbersMap["five"]) // null
//numbersMap.getValue("six") // exception!

val mutNumsMap = mutableMapOf("one" to 1, "two" to 2)
mutNumsMap.put("three", 3)
mutNumsMap.put("three", 4)
1110nullres5: kotlin.Int? = 3
```

Loops


```
for (i in 1..10) {
    print("$i ")
}
```

```
1 2 3 4 5 6 7 8 9 10
```

```
for(c in "hello") {
    print("$c ")
}
```

```
h e l l o
```

```
val names = listOf("eran", "moshe", "dave", "yael")
for (n in names)
    print("$n ")
```

```
eran moshe dave yael
```

downTo - inline function for descending order
step - for creating spaces

```
for(i in 20 downTo 1 step 2) {
    print("$i ")
}
```

```
20 18 16 14 12 10 8 6 4 2
```

Once int become Int we get these infix functions and much more
while, do-while - same as Java

Functions

fun - keyword for function declaration

```
fun [name](params) : [return value]
```

If no return type is specified then Unit (Java's void) is omitted - unless it is a single line function and the return value is inferred

```
fun max(a:Int,b:Int) : Int {
    return if(a>b) a else b
}
```

If the function is only one line no need for {} just use =

```
fun max(a:Int,b:Int) : Int = if(a>b) a else b
```

And in that case the return type can be inferred!

```
fun max(a:Int, b:Int) = if(a>b) a else b
```

vararg - for unknown number of params

```
fun max(vararg nums:Int) : Int {
    var max = Int.MIN_VALUE
    for(n in nums) {
        max = if(n > max) n else max
    }
    return max
}

println(max(5,8,3,20,7))
```

Move to IntelliJ IDEA create a new project, make sure your JDK is defined and in it create a new Kotlin file.

Kotlin enables top class variables and functions - no need for class to write functions

Kotlin also allows us to define inner functions - function nested inside other functions

```
fun main(args:Array<String>) {} [you can shorten by writing main followed by enter]
```

Print your first Hello World in Kotlin!!!

note: In Kotlin versions earlier than 1.3, the main function must have a parameter of type Array<String>.

Create infix functions

Functions marked with the infix keyword can also be called using the infix notation (omitting the dot and the parentheses for the call). Infix functions must meet the following requirements:

- They must be member functions or **extension functions** - they must have this.
- They must have a single parameter.
- The parameter must not **accept variable number of arguments** (varargs) and must have no **default value**.

Example (an Int extension function that concatenating a string to himself a given times)

```
infix fun Int.times(string: String) = string.repeat(n: this)
```

Usage:

```
print(7 times "Mush")
```

Infix notation also works on members functions (methods):

```
class Person(val name: String) {  
    private val likedPeople = mutableListOf<Person>()  
    infix fun likes(other: Person) { likedPeople.add(other) }  
}  
  
fun main() {  
    val sophia = Person(name: "Sophia")  
    val claudia = Person(name: "Claudia")  
    sophia likes claudia  
}
```

Please note: We will discuss classes later but for now notice that there is no new in Kotlin just the class name followed by it's constructor call.

Look at the smart cast we have seen before in IntelliJ and functions. Sometimes Kotlin programs need to work with null values, such as when interacting with external Java code or representing a truly absent state. Kotlin provides null tracking to elegantly deal with such situations - in the following example maybeString is smart casted to String

```
fun describeString(maybeString: String?): String {
    if (maybeString != null && maybeString.length > 0) {
        return "String of length ${maybeString.length}"
    } else {
        return "Empty or null string"
    }
}
```

Default arguments

Function parameters can have default values, which are used when you want to skip the corresponding argument. This significantly reduces the number of overloads and saves us ALLOT of code.

A default value is defined using = after the type.

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

Think about how many lines of code you just saved (please note that ByteArray is java array of bytes - it is not Array<Byte>)

```
val b = byteArrayOf()
read(b)
read(b, off: 100)
read(b, len = 100)
read(b, off: 100, len: 100)
```

If a default parameter precedes a parameter with no default value, the default value can only be used by calling the function with **named arguments**:

```
fun foo(  
    bar: Int = 0,  
    baz: Int,  
) { /*...*/ }  
  
foo(baz = 1) // The default value bar = 0 is used
```

We usually put the parameters with default values after parameter with no default values (no save for the need of using parameters names)

Please note that with default values we still have to write the type. The main reason is compilation performance. When we look at a method call, it helps a lot to have explicit parameter types which don't need to be inferred. Inferring the parameter type from a string constant is trivial, but if you have `fun foo(a = bar())` and `bar()` also has an inferred return type, understanding the actual parameter type of `a` becomes very expensive.

Drills

1. Ask the user for his name and his age and print out if he is old enough to drive (please use the `[String].toInt()` function but be aware that it works on `String` and not `String?` - what that the `readLine()` returns)

Spoilers:

`readLine()` returns `String?`

`toInt()` can be applied only to `String`

You must use `?.toInt()` Or `!!`

While `?.toInt()` can return null you can't use it in a boolean expression (can't compare null)

So the only option is to use `!!` but be sure to check before using

If you check before then you don't need `!!` Kotlin will automatically smart cast your object to `String` before applying the `toInt()` function

Solution:

```

fun main(args:Array<String>) {
    println("What is your name?")

    val name = readLine()
    println("What is your age?")

    val age = readLine()

    println(if(!age.isNullOrBlank() && age.toInt() > 18)
        "$name is old enough to drive"
        else "sorry but $name too young or i
}

```

2. Create a concat function that receives a list of strings and a separator and return one string containing the strings separated with the separator. If no separator supplied use comma. invoke her twice, once with the default separator and one with you own.

Solution:

```

println(concat(listOf("eran", "dfdf", "dfds")))
println(concat(listOf("eran", "dfdf", "dfds"), separator: "-"))
}

fun concat(strings : List<String>, separator : String = ",")
    = strings.joinToString(separator)

```

2 - Bonus - do you think you can supply both parameter but in a different order without changing the function title?

Solution(use parameters names):

```

val result1 = concat(separator = "-", list = listOf("eran", "moshe", "dave"))

```

3. Function syntax - Create the following functions:
 - A simple function that takes a parameter of type String and returns Unit.
 - A function that takes two strings message and a prefix. the second parameter is optional with default value "Info". The function will not return anything but this time use omitted Unit return value and print to the screen the prefix followed by the message.
 - A function that receives two integers and returns their sum.
 - A single-expression function that returns an integer (inferred) - the function will receives two Int and returns their multiplication.
 - A function that takes String varargs and prints them

- Infix function called "onto" that works on two strings (this and the parameter and return a new Pair containing both of them) - Pairs can be add to maps like we have seen before. use your infix function with map initialization.
- Create the main function and test all your functions

Solutions:

```
fun printMessage(message: String): Unit {
    println(message)
}

fun printMessageWithPrefix(message: String, prefix: String = "Info") {
    println("[$prefix] $message")
}

fun sum(x: Int, y: Int): Int {
    return x + y
}

fun multiply(x: Int, y: Int) = x * y
```

```
fun printAll(vararg messages: String) {
    for (m in messages) println(m)
}
printAll("Hello", "Hallo", "Salut", "Hola", "你好")
```

```
infix fun String.onto(other: String) = Pair(this, other)
val myPair = "McLaren" onto "Lucas"
```

For more reasons adding on basic functions syntax

<https://kotlinlang.org/docs/functions.html>

Exceptions

Kotlin solves a very common problem with try-catch block and variable scopes: In java we sometimes have to define a variable outside the try block and initialize it to null, we can't define it inside the try block because of the scope - if we define him in the try block then it wouldn't exist outside of it.

Kotlin solve this by with a try - catch block that is also an expression and thus return a value, the last line in the block is the returned value :

```
val result = try {
    write(byteArrayOf())
}
catch (ex : IOException) {
    print("An error")
    false
}finally {
    print("In finally")
}
print(result)
```

```
fun write(bytes: ByteArray?) : Boolean {
    if(bytes == null || bytes.isEmpty()) throw IOException()
    return true
}
```

Please note the we can throw IOException without surrounding it with try catch block or declare it in the method constructor!

This is not possible in Java cause IOException is a checked exception but **in Kotlin There are no checked exceptions!! like C# and Ruby** All is un-checked exceptions meaning it's your responsibility!

Thats why the keyword throws does not exists in Kotlin

Be responsible! Kotlin is more interested in saving us lines of code then in being our Mom and Dad :)

Kotlin Functional Programming

Download the full source codes from here

<https://drive.google.com/file/d/1JyVLjUWXQbmXIkUifjKlfzh-iipPNhv4/view?usp=sharing>

Kotlin is not a pure functional language but do support Lambdas and High order functions.

Lambdas - anonymous functions

Lambda expressions and anonymous functions are *function literals*. **Function literals are functions that are not declared but are passed immediately as an expression**

```
val addTwo: (Int) -> Int = { x -> x + 2 }  
println(addTwo(2))
```

We created an anonymous function and save her in a val later we will send it to another function. The function that receives it is will be called an **High Order function**.

Our Lambdas can be shorter using Type inferred - Kotlin deduced it from the function - but remember parameters types can never be inferred.

```
val sub = { x: Int, y: Int -> x - y }  
println(sub(5, 3))
```

We can use Lambdas in High order functions - functions that receives other functions as arguments

filter() - a pre-existing high order function that filtering a list using a given predicate. A Predicate is a lambda function that take a collection element and return a boolean value: true means that the given element matches the predicate, false means the opposite

```
val list = (1..100).toList()

println(list)//the whole list

println(list.filter { x -> x % 3 == 0 })//only the numbers divided by 3

println(list.filter { it % 4 == 0})//using implicit it
//default lambda parameter
```

1. Because the function receives Lambdas we can discard the () also if the lambda is the last parameter it can be written outside the parentheses
2. Implicit **it** - if the Lambda has only one parameter it can be implicitly called by **it**

According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses - Such syntax is also known as **trailing lambda**.

When using default parameters, If the last argument after default parameters is a **lambda**, you can pass it either as a named argument or **outside the parentheses**. If the lambda is the only mandatory argument in that call, the parentheses can be omitted entirely:

```
fun foo(
    bar: Int = 0,
    baz: Int = 1,
    qux: () -> Unit,
) { /*...*/ }

foo(1) { println("hello") } // Uses the default v
foo(qux = { println("hello") }) // Uses both default
foo { println("hello") } // Uses both default
```

You can explicitly return a value from the lambda using the return syntax. Otherwise, the value of the last expression is implicitly returned.

If the lambda parameter is unused, you can place an underscore instead of its name:

```
mapOf(1 to "h").forEach { (_, value) ->
    print("$value")
}
```

Function pointer ::

If we want to pass non-anonymous function we can use the :: which is a function pointer

```
fun isEven(x: Int) = x % 2 == 0

println(list.filter(::isEven))
```

The filter function will call the isEven function passing it each element on the list and will filter by it's result

map

Returns a list containing the results of applying the given [transform](#) function to each element in the original array

```
// val list = (1..100).toList()
val doubled = list.map { it * 2 }
println(doubled)

val avg = list.average()
val shifted = list.map { it + avg }
println(shifted)
```

```
val names = listOf("eran", "moshe", "raviv")
println(names.map { it.contains( other: "a" ) }) //output: [true, false, true]
```

In the last example we get a list of Boolean since it is the result of applying the contains function.

flatMap

Returns a single list of all elements yielded from results of [transform](#) function being invoked on each element of original array.

In other words both function applies a function to each of the elements but flat map also "FLATTEN" it - To understand what *flattening* a stream consists in, consider a structure like [[1,2,3],[4,5,6],[7,8,9]] which has "two levels".

Flattening this means transforming it in a "one level" structure :

```
[ 1,2,3,4,5,6,7,8,9 ]
```

```
val nested = listOf((1..10).toList(), (11..20).toList(), (21..30).toList());

println(nested)

val descendingList = nested.flatMap { it.sortedDescending() }
println(descendingList)

val descendingList1 = nested.flatMap { it }.sortedDescending()
println(descendingList1)

val descendingList2 = nested.flatten().sortedDescending()
println(descendingList2)
```

Please note that in the above example the `it` refers to the list inside - the `it` the inner object

More on collection sorting here (please read and note the `sortBy` function that also receives a selector)

<https://www.baeldung.com/kotlin/sort>

Let's use complex data like JSON objects.

Lets make a data structure of parents and their children and play a little

```
val data = listOf(mapOf("eran" to listOf(4,8,10)),
    mapOf("moshe" to listOf(6,8,9)),
    mapOf("Roni" to listOf(7,13,18)))

println(data) //output : [{eran=[4, 8, 10]}, {moshe=[6, 8, 9]}, {Roni=[7, 13, 18]}]
println(data.flatMap { it.values })//output: [[4, 8, 10], [6, 8, 9], [7, 13, 18]]

//Same output: [4, 8, 10, 6, 8, 9, 7, 13, 18]
println(data.flatMap { map -> map.flatMap { it.value } })
println(data.flatMap { it.values.flatten() })
println(data.flatMap { it.values }.flatten())
```

Drills

1. Using the data above Print one list that contain only the ages of 1 to 10

Solution:

```
println(data.flatMap { it.flatMap { it.value.filter { it <= 10 } }})
//or
println(data.flatMap { it.values.flatten() }.filter { it in 1..10 })
```

2. Define a map which contain each parent and his children ages (not list of maps, just one map) and put some faulty ages between the correct ones (like -6 or 150). Then print out the average children's age of each parent (take only the valid ages to the already exist Kotlin `average()`)

function) - use map this time. Try to solve this in two ways, use both lambdas and the :: (function pointer)

Solution:

```
val faultyData = mapOf(
    "eran" to listOf(2,6,8),
    "moshe" to listOf(8,150,19,-1,16),
    "ronit" to listOf(5,-6,10,15)
)

//val isValidAge:(Int)->Boolean = { it in 0..120 }
println(faultyData.map { it.value.filter { it in 0..120 }.average() })

//or
fun isValidAge(x:Int) = x in 0..120
val avg = faultyData.map { it.value.filter(::isValidAge).average() }
println(avg)
```

3. Now print the average of all the children together

Solution:

```
println(faultyData.flatMap { it.value.filter (::isValidAge) }.average())
//or
val avg1 = faultyData.flatMap { it.value }.filter { it in 0..120 }.average()
println(avg1)
```

4. Print the names of parents with faulty ages. Hint: search the Kotlin API for function that applies boolean condition on a list

Solution:

any() - Returns true if at least one element matches the given [predicate](#) (boolean values functions)

```
println(faultyData.filter { it.value.any { it !in 0..120 } }.keys)
```

5. Print the number of faulty ages

Solution:

```
val numberOfFaultyAges = faultyData.flatMap { it.value }.filter { it !in 0..120 }.size
```

6. Print only the names of the parents who has no faulty data (use all())

Solution:

```
val namesWithValidAges = faultyData.filter { it.value.all { it in 0..120 } }.keys
println(namesWithValidAges)
```

take(n) - takes only the first n elements

drop(n) - leaves the first n elements and leaves the rest

first(), last() - the first and last elements of the list

takeIf()

take the list if the the given predicate gives true

Take the list if it contains 4 if not we will get null

```
println(list.takeIf { it.contains(4) })
```

Note: takeIf() and takeUnless() is part of the Kotlin Standard library and will be mentioned later on. It can be applied to any instance of any class.

zip() - create list of Pairs. each pair will consist of key from the first list and value from the second. If one list is longer than the second it's extra values will be ignored

```
val names = listOf("eran", "moshe", "dave", "yael")
val numOfChildren = listOf(3, 2, 5, 3);

val zipped = names.zip(numOfChildren);
println(zipped)
```

Drill

Use zip function to create a list of pairs that each pair's key is the name and the value is true or false whether the name contains the character 'a'

```
println(names.zip(names.map { it.contains("a") })))
```

For more reading on קצ and High Order functions:

<https://kotlinlang.org/docs/lambdas.html#higher-order-functions>

Lazy Sequence

```
val longList = (1..9999999L).toList()
```

```
val time1 = measureTimeMillis {
    val sum = longList.filter { it > 50 }.map { it * 2 }.take(n: 1000).sum();
    println(sum)
}
println("time in millis $time1")
```

This is about 1-2 seconds

```
val time2 = measureTimeMillis {
    val sum = longList.asSequence().filter { it > 50 }.map { it * 2 }.take(n: 1000).sum();
    println(sum)
}
println("time in millis $time2")
```

This takes only 10 mili-seconds

Sequences are containers where the main difference between them and all the other collections is that the **actual computing is done only when needed** - in both examples we only needs the first 1000 not on all of the elements. In the first example the computations were made on all the list, but in the second one the computations were made on only the ones we needed - the first 1000 elements - this is the meaning of Lazy Sequences. And in general Lazy is a wide concept in Kotlin meaning that it's only done when needed.

Because of this Sequences can be infinite.

```
val seq = generateSequence(seed: 0) { it + 10 }

println(seq.take(n: 1000).toList())
```

If we don't take the first 1000 the program will go to infinite loop - he tries to turn infinite loop into list

Lazy Algorithm for finding prime numbers - Sieve of Eratosthenes

This algorithm is based on taking each number and removes all his multiplications

Drill - Advance

Build an algorithm based on the sieve of Eratosthenes that print the first n primes - use sequences

Solution:

```
val possiblePrimesAfter2 = generateSequence( seed: 3) {it + 2}
val primes = generateSequence( seed: 2 to possiblePrimesAfter2) { it: Pair<Int, Sequence<Int>>

    val p = it.second.first()

    val possiblePrimesAfterP = it.second.filter { it % p != 0}

    p to possiblePrimesAfterP ^generateSequence
}.map { it.first }

println(primes.take( n: 100).toList())
```

Scope functions

The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object. When you call such a function on an object with a **lambda expression** provided, it forms a temporary scope. In this scope, you can access the object without its name. Such functions are called *scope functions*. There are five of them: **let**, **run**, **with**, **apply**, and **also**.

Basically, these functions do the same: execute a block of code on an object. What's different is how this object becomes available inside the block and what is the result of the whole expression.

Here is a short guide for choosing scope functions depending on the intended purpose:

- Executing a lambda on non-null objects: **let**
- Introducing an expression as a variable in local scope: **let**
- Object configuration: **apply**
- Object configuration and computing the result: **run**
- Additional effects: **also**
- Grouping function calls on an object: **with**

here are two main differences between each scope function:

- The way to refer to the context object
- The return value.

The scope functions differ by the result they return:

- **apply** and **also** return the context object.
- **let**, **run**, and **with** return the lambda result.

The return value of **apply** and **also** is the context object itself. Hence, they can be included into call chains as *side steps*: you can continue chaining function calls on the same object after them. They also can be used in return statements of functions returning the context object.

let, **run**, and **with** return the lambda result. So, you can use them when

assigning the result to a variable, chaining operations on the result, and so on.

let()

define a scope a variable - it applies the the variable that we opened the scope on, we can give it a name

```
File( pathname: "text.txt").bufferedReader().let {reader ->

    if (reader.ready()) {

    }

}

//or

File( pathname: "text.txt").bufferedReader().let { it: BufferedReader

    if (it.ready()) {

    }

}
```

We can use **run()** and instead of the lambda argument (**it**) we have the **lambda receiver (this)**:

```
File( pathname: "text.txt").bufferedReader().run { this: BufferedReader

    if(ready()) {

    }

}
```

On the other hand, if this is omitted, it can be hard to distinguish between the receiver members and external objects or functions. So, having the context object as a receiver (**this**) is **recommended for lambdas that mainly operate on the object members: call its functions or assign properties.**

Drill

Create a list of names and print the size of only the names that are longer then 3 using let

```
val names1 = listOf("eran", "nir", "yael", "avi", "gershon")
names1.map { it.length }.filter { it > 3 }.let { it: List<Int>
    println(it)
}
//instead of
val names2 = names1.map { it.length }.filter { it > 3 }
print(names2)
```

If the code block contains a single function with it as an argument, you can use the method reference (::) instead of the lambda:

```
//or
names1.map { it.length }.filter { it > 3 }.let(::println)
```

We can use let for working with nulls - we enter the block only if it exists maybe aging str is not null

```
val str: String? = "eran"
//one option
//if(str.isNotEmpty()) forbidden because str can be null
//if(str?.isNotEmpty()) forbidden cause if accepts Boolean and not ?Boolean
//if(str!!.isNotEmpty()) Dangerous
//if(str!=null && str.isNotEmpty()) //the only option - use kotlin smart-cast
str?.let { it: String
    if (it.isNotEmpty()) {
    }
}
```

run()

run is used to execute a block of code and return the result - if the run function invoked on an object - not mandatory - unlike let - you can refer it by this (not it like let), like let it returns the lambda result

Combine let and run :

```
var p : String? = null
p?.let { println("p is $p") } ?: run { println("p was null. Setting default value to: ")
    p = "Kotlin"}

println(p)
//output:
//p was null. Setting default value to:
//Kotlin
```

also()

also is very similar to let but instead of the lambda result it **return the object itself** (both have it)

```
data class Person(var first: String, var last : String)
val person = Person("Moshik", "Afia")

val l = person.let { it.last = "Afik" }
val al = person.also { it.last = "Afik" }

println(l) //output: kotlin.Unit - the lambda return Unit
println(al) //output: Person(first=Moshik, last=Afik)
```

The also expression returns the data class object whereas the let expression returns nothing (Unit) as we didn't specify anything explicitly.

apply

The **context object** is available as a receiver (this). The **return value** is the object itself.

This is what differentiate it from also - Apply and also are almost the same apply has **this** and also it

```
person.apply { first = "Moshon" }

person.also { it.first = "Mushi" }
```

Use apply for code blocks that don't return a value and mainly operate on the members of the receiver object. The common case for apply is the object configuration. Such calls can be read as " *apply the following assignments to the object.*"

```
class Person(var name:String = "moshe", var age:Int = 7)

val p = Person().apply { this: Person
    name = "eran"
    age = 9
}
```

On the other hand, We should use also only when we don't want to shadow this.

with()

"with this object, do the following." - we have **this**

```
with(person)
{ this: Person
    first = "John"
    last = "Doe"
}
```

Because we have **this** it is very similar to apply but with 3 major difference:

1. Apply must work on an instance (the receiver) in with the instance is supplied as a parameter
2. with returns the lambda's result:

```
var check = with(person)
{ this: Person
    first = "John"
    last = "Doe"
    val xyz = "End of tutorial"
    xyz ^with
}

println(check) //End of tutorial
```

For more reading on scope functions:

<https://kotlinlang.org/docs/scope-functions.html>

Please Note that all of the scoped functioned mentioned above are part of the Kotlin's **standard functions**

There you can also find a function we already discussed **takelf()** & **takeUnless()**

Lets look closely on takelf:

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

From it, we notice that

- It is called from the T object itself. i.e. T.takeIf.
- The predicate function takes T object as parameter
- It returns this or null pending on the predicate evaluation.

Thus it is very useful in null checks:

```
// Original code
if (someObject != null && status) {
    doThis()
}

// Improved code
someObject?.takeIf{ status }?.apply{ doThis() }
```

You can read more on takeIf() from where is example is [taken from](#)

use()

Executes the given [block](#) function on this resource and then closes it down correctly whether an exception is thrown or not.

Must be used on objects that implements the java closable interface

```
FileReader( fileName: "fdfsd").use { it: FileReader
    it.read()
}
```

No need to close the FileReader

Inline function

All of our scoped functions were **inline**

Meaning the compiler copies the function code to the place where the we invoke it

When using lambda the compiler create an instance of the function each time unless we declare the function be inline and then he just copies the function code

```
var s = "moshe"  
s = modify(s) { it.toUpperCase() }  
println(s)  
}
```

```
inline fun modify(str:String, operation:(String)->String) = operation(str)
```

If we add the keyword **noinline** before the lambda the compiler will alert us that the inline keyword has no meaning cause it saves time when working with functions

```
inline fun modify(str:String, noinline operation:(String)->String) = operation(str)  
inf Expected performance impact from inlining is insignificant. Inlining works best for functions  
with parameters of functional types
```

Operator Overloading in Kotlin

Use **operator** keyword before the fun

We already have by default the == and != when we override the Any's equals() method

We can overload the plus, minus, times, unsayMinus, unaryPlus, inc(++), dec(--), and not

Kotlin has a special treatment of Java's Comparable.

Simply put, we can call the compareTo method in the Comparable interface by a few Kotlin conventions. In fact, any comparisons made by "<", "<=", ">", or ">=" would be translated to a compareTo function call.

In other words, If we want to overload the > , >= and < , <= we need to implement the Comparable interface

```
data class Point(var x:Int, var y:Int) : Comparable<Point> {  
  
    operator fun plus(other:Point) = Point(x: x+other.x, y: y+other.y)  
  
    operator fun minus(other:Point) = Point(x: x-other.x, y: y-other.y)  
  
    operator fun times(other:Point) = Point(x: x*other.x, y: y*other.y)  
  
    operator fun unaryMinus() = Point(-x, -y)  
  
    operator fun inc() = Point(x: x+1, y: y+1)  
    operator fun not() = Point(y, x)  
  
    override fun compareTo(other: Point): Int {  
        return ((x+y)/2.0 - (other.x+other.y)/2.0).toInt()  
    }  
}
```

```
fun main(args: Array<String>) {  
  
    val p1 = Point( x: 1, y: 2)  
    val p2 = Point( x: 2, y: 1)  
  
    println(p1 >= p2)  
    val p3 = p1 + p2;  
    println(p3)  
  
    var p4 = p1 - p2;  
    println(p4)  
  
    println(-p4)  
  
    println(p4++)  
    println(p4)  
  
    println(p3)  
    println(p1)  
  
    println(p3*p1)  
  
    val p5 = !(p3*p1)  
    println(p5)  
}
```


Kotlin OOP

Download the full Chapter source code from here:

<https://drive.google.com/file/d/1tpnZIXR7vaBfdJFVjOEw7x4BoRubKFjk/view?usp=sharing>

In Kotlin all objects inherit from class Any (it has toString, equals & hashCode)
Unlike Java in Kotlin we can combine few public classes in one file

Lets make our first class:

```
class Student()

fun main() {
    val s = Student()
}
```

Class student has an empty constructor
s has only the functions from Any

If we want the java setter we set it's properties as var if we want only getters they will be val

init function is called in any instance creation - no matter which constructor we invoked

```
class Student(first:String, age:Int) {

    var first:String
    var age:Int

    init {
        println("init called")
        this.first = first
        this.age = age
    }
}

fun main() {
    val s = Student( first: "moshe", age: 67)
    println("name ${s.first} age ${s.age}")
}
```

Please notice that when printing s.first and s.age we use {} meaning it's actually

a function invocation! Calling the get...

The . Actually invoke the getter that was auto generated

Same as s.age = 80; -> this only ok if age is var and not val because the setter is invoked

These are Properties

A better way for writing the class - Kotlin way

```
class Student(first:String, age:Int) {
    var first = first
    var age = age

    init {
        println("init called")
    }
}
```

The init is still called before the initialization

A better way then this (show with auto correction)

```
class Student(var first: String, var age: Int)
    init {
        println("init called")
    }
}
```

This is the primary constructor!

When we add the var or val in the **constructor** then we set the properties to the parameters passed to the constructor and there will be default getters and setters according to the var or val. That is why we see classes that are just one line with no body

```
class Student(var first: String, var age: Int)
```

Think of how coding you just saved!

If we want our custom getter and setter we can't declare the properties in the constructor but in the class body and provide a custom get() and set(value) functions

field - reference for the actual value

```
class Student(first:String, age:Int) {
    var first = first
    get() = "name is $field"

    var age = if(age>0) age else 0 //for the ctor initialization
    set(value) {
        if(value > 0)
            field = value
    }
}
```

Because of default parameters we usually don't need ctor overloading

```
class Student(var first: String, age: Int = 25) {
```

```
val s1 = Student( first: "Rona")
```

If age is not passed to the ctor then he will get the default value

Please note that If you supply one value, it's used for the first named parameter (you can use parameters name to overcome this) so it generally doesn't make any sense to provide a default value for an early parameter without providing a default for subsequent parameters.

If you're not going to provide default values for all parameters, you should only provide default values for the last parameters in the constructor:

```
class Student(var first: String = "moshe", age: Int) {
    fun main() {
        val s = Student( first: 25)
    }
}
```

The integer literal does not conform to the expected type String

In this case you need to specify the second parameter by his name

```
val student = Student(age = 23)
```

If for some reason we want the java way or you want a whole new constructor all together you can create another one it is called secondary constructor. We can do it using the **constructor** keyword (in the primary ctor the keyword is deferred). We can delegate to the primary constructor

```
class Student(val first:String, age:Int = 20) {

    var age = if(age>0) age else 0 //when passing negative to the ctor
    set(value) { //when using the setter
        if(value > 0)
            field = value
    }

    constructor(first: String) : this(first, age: 40) {
        //Do something here
    }
}

fun main() {
    val s1 = Student( first: "mika")
    println(s1.age) //40!
```

In this case if we create a Student with name alone he will be 40 and not 20! The system will always look for the exact constructor before applying default values!

private constructor

To avoid the public constructor in cases such as singletons add the **private constructor()** after the class declaration or inside the class body - better as primary constructor (in the class declaration)

```
class Student private constructor(val first:String, age:Int = 20) {
```

```
val s = Student( first: "Rona", age: 28)
```

Cannot access '<init>': it is private in 'Student'

lateinit var

Some variables needs to initialize later we can use the **lateinit var**

```
lateinit var last:String
```

In this case the compiler won't show the compilation error of the var not being initialize BUT be careful from accessing the variable -

UninitializedPropertyAccessException will be thrown at runtime

Only lateinit var exist not val (the lateinit gives it some initial value) and it's not working on all the Java primitives(Int, Double, Char...).

```
lateinit var grade : Int
// 'lateinit' modifier is not allowed on properties of primitive types
```

Delegated properties

Another way of late initialization is using delegated properties: we will create an object that when we first access it's properties the delegate object is created and store the value computed in the object. This can also work on java primitives.

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

The expression after **by** is a *delegate*. The `get()` and `set()` corresponding to the property that will be delegated to its `getValue()` and `setValue()` methods. Property delegates doesn't have to implement any interface, but they have to provide a `getValue()` function (and `setValue()`--- for `var` s).

If we are want `val` or one the java primitives to get a later value we can use this delegate:

```
val grade by Delegates.notNull<Int>()
```

But again be careful from accessing it before initialization
Take a look at his code:

```
public object Delegates {
    Returns a property delegate for a read/write property with a non-null value that is initialized r
    during object construction time but at a later time. Trying to read the property before the initia
    value has been assigned results in an exception.
    Samples: samples.properties.Delegates.notNullDelegate
    // Unresolved
    public fun <T : Any> notNull(): ReadWriteProperty<Any?, T> = NotNullVar()
```

```
private class NotNullVar<T : Any>() : ReadWriteProperty<Any?, T> {
    private var value: T? = null
    public override fun getValue(thisRef: Any?, property: KProperty<*>): T {
        return value ?: throw IllegalStateException("Property ${property.name} should be initialized before get.")
    }
    public override fun setValue(thisRef: Any?, property: KProperty<*>, value: T) {
        this.value = value
    }
}
```

Another option is using the lazy function

The lazy function will be invoked only when the object id accessed for the fist time

```
val last by lazy {
    println("lazy")
    "Cohen" ^lazy
}
```

The println is used to show you that the initialization is happening only when first accessed

Another example in using observable delegates

Delegates.observable - Returns a property delegate for a read/write property that calls a specified callback function when changed.

```
class User {
    var name : String by Delegates.observable( initialValue: "moshe" ) {
        _,old,new ->
        println("$old -> $new")
    }
}

fun main() {
    val user = User()
    user.name = "rona" //outputs: moshe -> rona
    user.name = "dave" //outputs: rona -> dave
}
```

If you want to intercept assignments and veto them, use `vetoable()` instead of `observable()`. The handler passed to the vetoable is called *before* the assignment of a new property value.

```
var last by Delegates.vetoable( initialValue: "moshe" ) {
    _,old,new ->
    println("$old $new")
    false ^vetoable
}
```

You read more about the concept of Delegates and maybe create your own here:

<https://kotlinlang.org/docs/delegated-properties.html>

And of course on in this course we will have our own delegate that changes the property value according the the attached fragment lifecycle.

Class extensions

We can add functions to existing classes in Kotlin using the class name before the function name.

Inside the function we have **this**

```
fun Int.isEven() = this % 2 == 0
val x = 4;
println(x.isEven())
```

```
fun Student.isOldEnough() = this.age > 25

val s = Student( first: "mush", age: 89)
println(s.isOldEnough())
```

Data class

```
data class Contact(var name:String, var email:String, var id:Long)
```

Using the keyword **data** we can use Kotlin to create everything needed for a data class - equals, hash-code, toString, copy and more (a Whole file in java is just one line)

```
val moshe = Contact( name: "Moshe Cohen", email: "moshe@moshe.com", id: 1234567)

println(moshe) //calls toString

val moshe1 = Contact( name: "Moshe Cohen", email: "moshe@moshe.com", id: 1234567)

println(moshe == moshe1) //calls equals return true
println(moshe === moshe1) //reference checking - return false

val moshon = moshe.copy(name = "mush") //copy the object with new name

println(moshon)
```

In Data classes a componentN function is created for each the the properties. We can use this for destructuring declaration

Destructuring declarations

A destructuring declaration creates multiple variables at once(only local variables).

```

val(name,email) = moshe
// same as
val name1 = moshe.component1()
val email1 = moshe.component2()
//the component functions created because of the data class
println("$name $email")

```

If you don't need a variable in the destructuring declaration, you can place an underscore instead of its name:

```

val(name,_,id) = moshe
//the component functions is not called when using _
println("$name $id")

```

The restructuring declaration can also help us in a variety for other things, for example when iterating on a map objects

```

val map = mapOf(1 to "a", 2 to "b")
for ((key,value) in map) {
    println("$key $value")
}

```

They are also used for function calls that we want to return more than one value and no need for the wrapper object

```

//inside main function cause destructuring declaration only allowed on local variables
val(result,status) = something()
}

fun something() : Result {
    return Result(result: 4, status: true)
}

data class Result(val result:Int, val status:Boolean)

```

Inheritance

The derived class doesn't need to add val or var to arguments already defined in the parent class.

Inheritance is defined by :

All Kotlin classes are final by default! This is mainly because almost no one wrote final in java

When we want to inherit from a class it must be declared as **open**

All the function that we want to override must be **open**


```

open class Shape1(val name:String) {
    open fun area() = 0.0
}

class Circle1(name: String, val radius:Double) : Shape1(name){

    override fun area() = Math.PI * Math.pow(radius,2.0)

    fun fill() = "Circle filled"
}

fun main(args:Array<String>) {

    val s : Shape1 = Circle1( name: "circle", radius: 7.9)
    println(s.area())
    //s.fill() can't find function fill in reference of type Shape

```

In Kotlin we can also override the **class properties** in cases where we want to add a setter for the child or write a different get and set functions.

If the property is val in the parent class in the derived class it can be either var or val - just add a setter - but if it var in parent class it can't be val in the child - we can't vanish the setter it already has one.


This is an error - we can't hide the parent name but we can override it and add a setter and a different default value

```

open class Shape (val name:String = "SHAPE") {}

class Circle(var name:String = "CIRCLE") : Shape(name) {

```

 'name' hides member of supertype 'Shape' and needs 'override' modifier

```

open class Shape (open val name:String = "SHAPE") {}

class Circle(override var name:String = "CIRCLE") : Shape(name) {}

```

We can also change the parents getter function:

```

val shape = Shape()
println(shape.name)//Shape SHAPE
val circle = Circle()
println(circle.name)//Circle CIRCLE
}
open class Shape (name:String = "SHAPE") {
    open val name = name
        get() = "Shape $field"
}
class Circle(name:String = "CIRCLE") : Shape() {
    override var name = name
        get() = "Circle $field"
}

```

Overriding methods always use the same default parameter values as the base method.

When overriding a method that has default parameter values, the default parameter values must be omitted from the signature

```

open class Shape(open val name:String = "SHAPE") {
    open fun add(x : Int = 6) {}
}
class Circle(override val name: String = "CIRCLE",val radius:Double) : Shape(name) {
    override fun add(x: Int = 10) {
        super.add(x)
    }
}

```

An overriding function is not allowed to specify default values for its parameters
[Remove default parameter value](#) [More actions...](#)

It make more sense for Shape to be **abstract** - the area function needs to be abstract

No need for **open** when using the **abstract** keyword same in functions or in classes

```

abstract class Shape1(open val name:String = "SHAPE") {
    abstract fun area() : Double
}

```

Casting(is, as and as?)

Kotlin can smart-cast our objects if we check them before using **is** -Note that smart casts work only when the compiler can guarantee that the variable won't

change between the check and the usage so it always works on val but only on var **local** properties(the compiler can track the local variables)

```
abstract class Shape(open val name:String = "SHAPE") {}

class Circle(override val name: String = "CIRCLE",val radius:Double) : Shape(name){
    fun roll() = println("I roll the circle")
}

fun main(args: Array<String>) {

    val shape:Shape = Circle( name: "Circle 1", radius: 5.6)
    if(shape is Circle) shape.roll()
}
```

If we want to cast the object ourselves we can use the unsafe **as** operator - Usually, the cast operator throws an exception if the cast isn't possible. And so, it's called *unsafe*. The unsafe cast in Kotlin is done by the infix operator **as**.

```
val c = shape as Circle
```

Or if you want to avoid exceptions, use the *safe* cast operator **as?**, which returns null on failure.

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

For more reasons adding on Type check and castings

<https://kotlinlang.org/docs/typecasts.html#type-erasure-and-generic-type-checks>

Interfaces

```
class Circle1(val radius:Double,override val name: String = "CIRCLE") : Shape1(name),Fillable1 {

    override fun fill() {} //must
    override fun area() = Math.PI * Math.pow(radius,2.0)
}

interface Fillable1 {
    fun fill()
}
```

Until now it was the same as Java but:

Interfaces in Kotlin can contain properties (without initialization) but when implementing it we must override it

```
class Circle1(val radius:Double,override val name: String = "CIRCLE") : Shape1(name),Fillable1 {
    override val color: String = "Black"
    override fun fill() = "Filling the circle with color $color"
    override fun area() = Math.PI * Math.pow(radius,2.0)
}

interface Fillable1 {
    val color:String
    fun fill():String
}
```

We can also define a default function implementation and in that case we don't have to override it. But that can cause the diamond problem and we solve it using the <[which]>

This is ok

```
class Circle1(val radius:Double,override val name: String = "CIRCLE") : Shape1(name),Fillable1 {
    override val color: String = "Black"
    // override fun fill() = "Filling the circle with color $color"
    override fun area() = Math.PI * Math.pow(radius,2.0)
}

interface Fillable1 {
    val color:String
    fun fill() = "Filling the fillable"
}
```

But if Shape has fill and also there is a default implementation in the interface which fill will be called and we solve it using <[parent]>

```
abstract class Shape1(open val name:String = "SHAPE") {
    abstract fun area() : Double
    open fun fill() = "Filling the shape"
}

class Circle1(val radius:Double,override val name: String = "CIRCLE") : Shape1(name),Fillable1 {
    override val color: String = "Black"
    override fun fill() : String {
        //super.fill() - error!
        println(super<Shape1>.fill())
        println(super<Fillable1>.fill())
        return "Filling the circle with color $color"
    }
    override fun area() = Math.PI * Math.pow(radius,2.0)
}

interface Fillable1 {
    val color:String
    fun fill() = "Filling the fillable"
}

fun main(args:Array<String>) {
    val c = Circle1( radius: 6.7)
    println(c.fill())
}
```

Object Expressions and Declarations

Object expressions create objects of anonymous classes, that is, classes that aren't explicitly declared with the class declaration - with a specific name. Such classes are useful for one-time use. You can define them from scratch, inherit from existing classes, or implement interfaces. Instances of anonymous classes are also called *anonymous objects* because they are defined by an expression, not a name - **Anonymous inner classes**

```
val w = Window.getWindows()[0]

w.addMouseListener(object : MouseListener {
    //need to implement the mouse listener functions
})
```

Like in java Anonymous class can access outside class members

```
var x = 0;
val w = Window.getWindows()[0]

w.addMouseListener(object : MouseListener {
    //need to implement the mouse listener functions

    override fun mouseClicked(e: MouseEvent?) {
        x++
    }
})
```

N

When only one function exist in the interface (SAM - Single Abstract Method) we will use Lambda and not object expression(will be discussed later on).

Another simple example :

```
val hello = object {  
    val hello = "Hello"  
    val world = "World"  
    //object expression also extends Any  
    override fun toString(): String {  
        return "$hello $world"  
    }  
}  
  
fun main() {  
    println(hello)  
}
```

Object declarations

Object declaration always has a name following the **object** keyword. Just like a variable declaration, an object declaration is not an expression, and it cannot be used on the right-hand side of an assignment statement.

The initialization of an object declaration is thread-safe and done on **first access**. To refer to the object, use its name directly.

Kotlin makes it easy to declare singletons using object declaration:

```
val car = CarFactory.create("Mazda")
val car1 = CarFactory.create("Toyota")
println(car)
println(car1)
println(CarFactory.numOfCars)
}

data class Car(val name:String, val price:Int)

object CarFactory {

    var numOfCars = 0

    fun create(name: String) : Car {
        numOfCars++
        return Car(name, price: 10000)
    }
}
```

Please note the because this is an object declaration it has no constructor and there is only one instance so it already a singleton!

```
val c = CarFactory()
Unresolved reference.
```

However, this is ok (this is just giving the object a different reference):

```
val fact = CarFactory
println(fact.numOfCars)
```

Such objects can have super-types and we can create a non anonymous single implementation

```

fun main(args: Array<String>) {

    // val c = CarFactory.createCar("Zoe")
    val w = Window.getWindows()[0]
    w.addMouseListener(MyMouseListener)
}

object MyMouseListener : MouseListener {
    override fun mouseClicked(e: MouseEvent?) {
        TODO(reason: "Not yet implemented")
    }
}

```

Companion objects

An object declaration inside a class can be marked with the companion keyword

This replaces the java statics.

Only one instance of the companion object is created for all instances of the class - it is an object - one per class like the static initializer - when we load the class to the memory for the first time then the companion object is created for all the upcoming instances that will share it.

Please note that it must be an object and inside it we will declare both properties and functions

```

class Student(var first: String = "moshe", age: Int) {

    companion object {
        var numOfStuednts = 0
        fun getAvgStudent() = Student( first: "dave", age: 25)
    }

    init {
        numOfStuednts++
    }
}

```

```

println("The average student is ${Student.getAvgStudent()} " +
        "and the num of students is ${Student.numOfStudent}")

```

The default companion object name is Companion but we don't need to specify

it just use the class name, you can also give the companion object a name but that is truly unnecessary

```
companion object Factory{  
    var numOfStuednts = 0  
    fun getAvgStudent() = Student( first: "dave", age: 25)  
}
```

If you have only one companion object you can still access it with the class name. And since each class is allowed only one companion object the naming is quite unnecessary and use it only if it makes your code more organized.

If you need the companion object itself just use the class name (or if it has a name - his name)

```
val comp = Student|  
comp.numOfStuednts
```

Note that even though the members of companion objects look like static members in other languages, at runtime those are still instance members of real objects, and can, for example, implement interfaces:

```
interface Factory<T> {  
    fun create() : T  
}  
  
companion object : Factory<Student>{  
    var numOfStuednts = 0  
    fun getAvgStudent() = Student( first: "dave", age: 25)  
    override fun create(): Student {  
        TODO( reason: "Not yet implemented")  
    }  
}
```

However, on the JVM you can have members of companion objects generated as real static methods and fields if you use the `@JvmStatic` annotation. See the [Java interoperability](#) section for more detail.

There is one important semantic difference between object expressions and object declarations:

- Object expressions are executed (and initialized) *immediately*, where they are used.
- Object declarations are initialized *lazily*, when accessed for the first

time.

- A companion object is initialized when the corresponding class is loaded (resolved) that matches the semantics of a Java static initializer.

For more reading:

<https://kotlinlang.org/docs/object-declarations.html#using-anonymous-objects-as-return-and-value-types>

Functional (SAM) interfaces

An interface with only one abstract method is called a *functional interface*, or a *Single Abstract Method (SAM) interface*. The SAM interface can only have one abstract method. To declare SAM interface use the keyword `fun` before the interface

The main advantage:

Instead of creating a class that implements a functional interface manually, you can use a lambda expression. With a SAM conversion, Kotlin can convert any lambda expression whose signature matches the signature of the interface's single method into the code, or

For example

Take the following interface:

```
fun interface IntPredicate {  
    fun accept(i: Int): Boolean  
}
```

If you don't use a SAM conversion, you will need to write code like this:

// Creating an instance of a class

```
val isEven = object : IntPredicate {  
    override fun accept(i: Int): Boolean {  
        return i % 2 == 0  
    }  
}
```

By leveraging Kotlin's SAM conversion, you can write the following equivalent code instead:

// Creating an instance using lambda

```
val isEvenSAM = IntPredicate { it % 2 == 0 }
```

And in main function:

```
println(isEvenSAM.accept(i: 7))
```

You will use this allot in Android programming just think of the OnClickListener interface, isn't it SAM?

Nested and inner classes

In Kotlin like in Java we can nest a class within another class (this is simply a structural thing):

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

Note that Nested can't access bar.

You can make a nested class Inner using the **inner** keyword before the class declaration.

A nested class marked as inner can access the members of its outer class!

Inner classes carry a reference to an object of an outer class. In nested class we don't need to create an instance of the outer class just use it's name (like we said before it's a structural thing). But if it is an inner class we must create an instance of the outer class to get and instance of the Inner class:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

Access Modifiers

private - same as java

public - the default in Kotlin!

protected - same as java

internal - equivalent to java package level - same Module - a set of Kotlin files compiling together - In android same Gradle or Maven

We can use **as** keyword for direct name for imported classes

```
import java.awt.Window as W
import java.awt.event.MouseEvent
import java.awt.event.MouseListener

fun main(args: Array<String>) {

    // val c = CarFactory.createCar("Zoe")
    val w = W.getWindows()[0]
```

Generics - covariance and contravariance

In case we want to define a generic class and later narrow the generic type (we used ? Extend Object in java) this is a problem since we set a more specific type of the generic, and we can add things at compile time that will crash at runtime!

```
class Source<T>()
fun main(args: Array<String>) {
    val source:Source<Any> = Source<String>()
    // val sources1:Source<String> = S
}
```

Type mismatch.
 Required: Source<Any>
 Found: Source<String>

The problem is that in the reference he accepts wider objects then in runtime

We can use the **out** keyword next to the generic to specify the T will only be used as return value - and the problem solved - covariance - because if it will be used only as a return value then the user won't be able to cause the problem mentioned before.

```
class Source<out T>()
{
    fun main(args: Array<String>) {
        val source:Source<Any> = Source<String>()
    }
}
```

Same in the opposite direction - we can specify the keyword **in** for generics that will only be used as parameter - think of a getter that suppose to return String but actually return Any - this is a problem! but think of a setter that suppose to get String and get Any - this is no problem - contra variance

```
class Source<in T>()
{
    fun main(args: Array<String>) {
        val source:Source<String> = Source<Any>()
    }
}
```

Its also possible to define both:

```
class Source<in T, out E>()
{
    fun main(args: Array<String>) {
        val source:Source<String,Any> = Source<Any,String>()
    }
}
```

sealed

Sealed classes and interfaces represent restricted class hierarchies that provide more control over inheritance. All direct subclasses of a sealed class are known at compile time. No other subclasses may appear after a module with the sealed class is compiled. For example, third-party clients can't extend your sealed class in their code. Thus, each instance of a sealed class has a type from a limited set that is known when this class is compiled.

This is very useful when checking instances of a certain class with `when()` because first if we chocked all known subclasses then we don't need else and more then that the compiler warnings that tells us that we forget to check a certain subclass can save us allot debugging time.

A sealed class is **abstract** by itself, it cannot be instantiated directly and can have abstract members.

Direct subclasses of sealed classes and interfaces must be declared in the same package.

Getting started

Download Android Studio Guide fore here:

<https://drive.google.com/file/d/1AGCDM9cX2J0vfGKW1Q1O--iKACx4RxeM/view?usp=sharing>

Download the full UI Guide from here

<https://drive.google.com/file/d/1lIB4jGart0MifC6kkgSExKFti3EeHxv-/view?usp=sharing>

Download the Full app created from the guide

https://drive.google.com/file/d/16x48WtHTlruxEHbv8EKA_Dze6PSQxPpR/view?usp=sharing

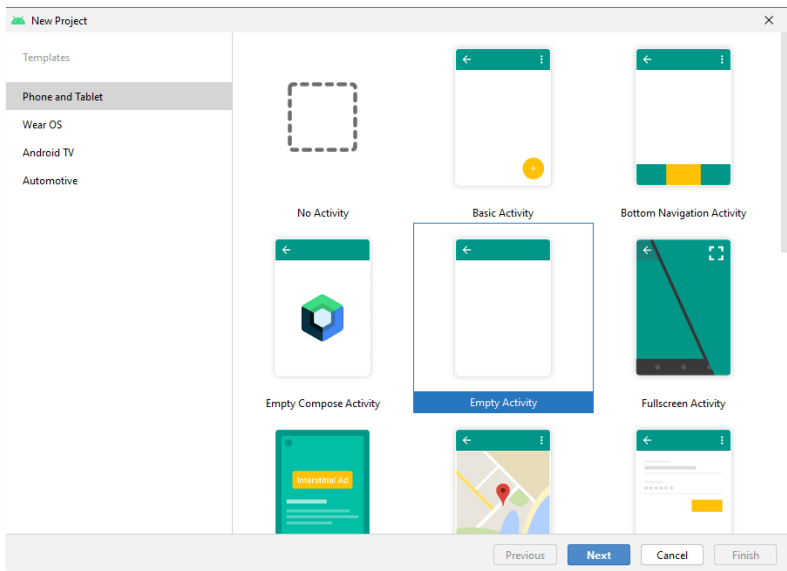
Download the picture archive file

<https://drive.google.com/file/d/1N4MXLjFKFGPy1kkUhFDqhGBi5sSP6vlf/view?usp=sharing>

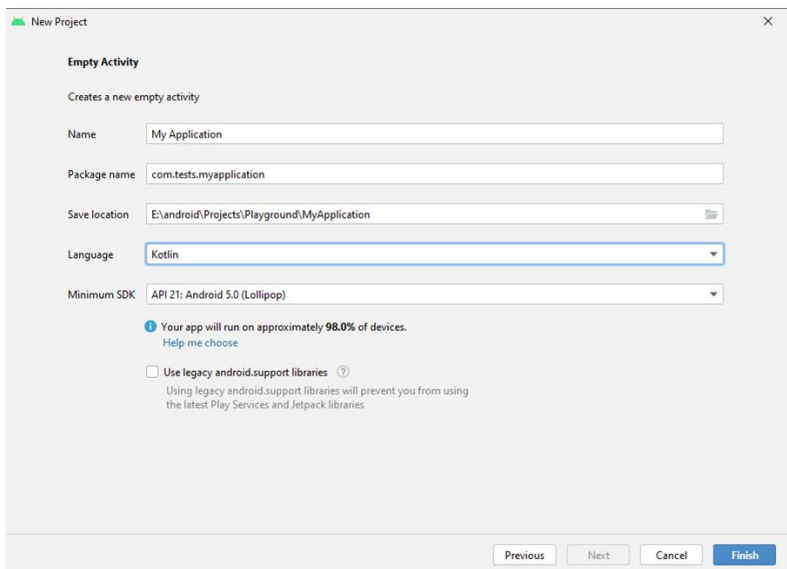
Open a new project in Android Studio

In this tutorial we will create a new project in Android Studio. Creating a new project is very simple. First, click on **File → New → New Project**. A new project dialog will pop up, with various templates. Android Studio provides many templates that we can choose from. Each template contains pre-generated code that is often used when creating a new project with specific components.

For now, we will create a new project containing a single Empty Activity.



After Clicking Next, we need to provide initial configuration settings.



Name – The name of the application.

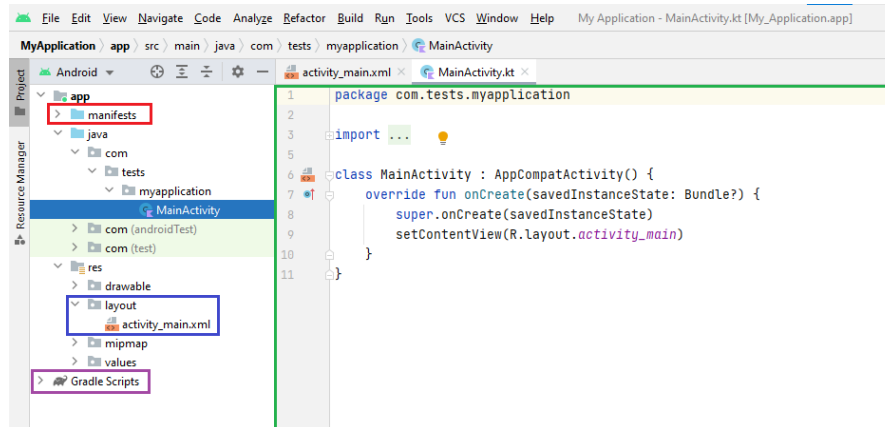
Package name – The name of the application root package name.

Save location – Local directory of the project.

Language - Project language.

Minimum SDK – The minimum SDK version of the Android operating system required to run the application.

After setting initial settings, click **Finish**. After Some processing, The Android Studio main window appears.

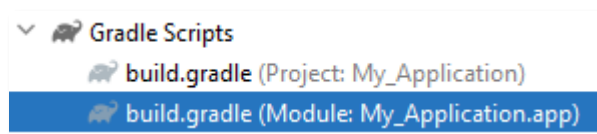


The MainActivity is the entry point for the application. When we build and run the app, the system launches an instance of this activity and loads its layout set with `setContentView()`.

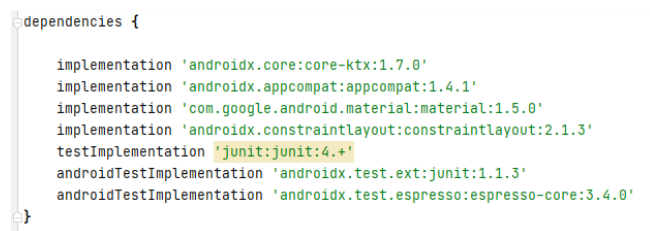
The Manifest file describes the basic characteristics of the app and defines each of its components.

The XML file defines the layout for the activity's UI. The default activity_main.xml file will contain a simple **TextView** with the text "Hello, World!".

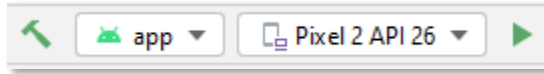
The gradle contains two gradle files: one for the project, and one for the app module. Each module has its own `build.gradle` file.



The app's gradle contains the SDK versions the app is using, compile options, Plugin declarations, and more. If we need to integrate a dependency into our app, we specify it in the "dependencies" block.



In order to run the app, click on the play icon in the top bar.



In the image above, an emulated device is set to run the app. If you have a physical device connected, it will show the device in the device window. **Make sure to enable USB debugging on the device:**

1. Open the device's settings.
2. Search for the developer options. If you don't see it, go to the **About** section in the settings, and tap the **build** number until a message appears saying you have access to developer options.
3. Enter the developer options, and scroll to the "Enable USB debugging". Enable the USB debugging. This will let us connect our device to the Android studio and run our app on the device.

Android User Interface Introduction

In this tutorial we will go over the different views that can be used inside an Android app. Understanding views and how to interact with them is a MUST for every android developer.

So what is a view?

From a user perspective, views are the visible stuff on the screen inside an app. Images , texts, buttons, cards and more, are all views. A View occupies a rectangular area on the screen and is responsible for drawing itself and handle its events such as touch events, clicks, swipes, state and more.

The `View` class itself is the base class for all views. Not all views inherit directly from it, some of them inherit from a different view. But because the `View` class is the base class, all views share a lot of functionality and attributes, but have their own special attributes and methods.

Some commonly used views (may be referred to as **widgets**) in applications are:

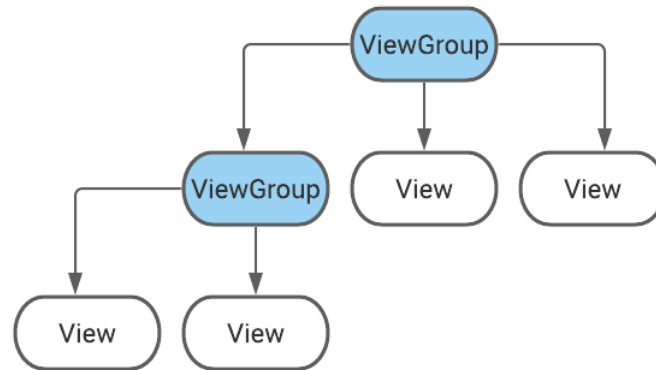
- `EditText` – lets the user input some text.
- `TextView` – provides an area in which we can insert some text (mostly used for none-editable text)
- `Buttons` – provide users with a clickable widget.
- `ImageButton` – Displays a button with an image that can be clicked.
- `ImageView` – provides an area in which we can insert an image.
- `SeekBar` – provides a visible and draggable indicator to the progress of an operation (like the YouTube progress bar in a video)

All views can be made clickable, Not just buttons (and also not clickable). Most views may be transformed in terms of content, visibility, size and animation. There aren't a ton of different types of views, but each view has its own attributes and functionality that differentiate it from other ones, and make it suitable for what we want to achieve.

Some views may contain other views, meaning a view can have children, and a parent. This type of view is called a **ViewGroup**, and it inherits from the View class.

ViewGroup

A `ViewGroup` is a subclass of the `View` class. It provides a space (invisible or painted) to hold other `ViewGroups` and views. Inside it, we can arrange the views by setting up their attributes, including values for their position inside the `ViewGroup`.



ViewGroups can contain other viewGroups and views

In order to contain other views, we don't use the `ViewGroup` class directly, but use a `Layout` class that inherits from it.

Some commonly used layouts in applications are:

- **LinearLayout** – Sets the views inside it one after the other - horizontally or vertically.
- **ConstraintLayout** – Sets the views inside it by constraining them to a specific view or its parent.
- **RelativeLayout** – Sets the views inside it relative to other views.
- **RadioGroup** – A class that inherits from `LinearLayout`, and can contain `RadioButton` views



Two `RadioButton` views inside a `RadioGroup`. `RadioGroup` allows us to control a group of radio buttons, and make sure only one is checked

Note: The way the layout draws itself is by traversing the view tree 2 times. On the first pass it **measures** the view tree recursively, calculating the dimensions of all the views. On the second pass, each view **layouts** all of its sub views according to their calculated size from the first pass.

Before we go further about View and ViewGroup, let's go over some basic measure units and constants that help us work with the different views.

DP vs SP vs PX

- **DP** – Density Independent Pixels – This unit is based on the physical density of the screen. 1 DP is roughly equal to 1 pixel. This ratio will change with the screen density, but not necessarily in direct proportion. Using this unit makes the view size inside the layout resize properly for different screen densities.
- **SP** – Scale Independent Pixels – This unit behaves like the DP unit, but it is also scaled by the user's font size settings. So for controlling the size of text, we use this unit, and use the DP unit for everything else.
- **PX** – Pixels – Actual pixels on the screen. This unit is not recommended because screen resolution and size change from device to device, so on one device our view may seem fine, but on another device – completely messed up.

Padding vs Margin

- **Padding** – The space between a view's content and its border.
- **Margin** - The space between a view's border, its parent layout and other views.

We can set these properties for a specific side of the view (bottom, up, left, right), or apply them on all the sides. Both of these attributes should be in DP units. If you choose to apply on one side, it's best practice to apply it on the other side as well (up and down, left and right). This is in order to keep everything symmetric, and support a Right-To-Left direction (explained later on).



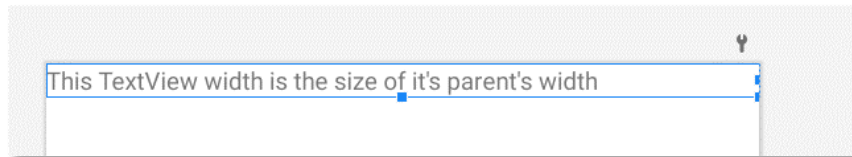
■ - margin ■ - padding ■ - view border

Note: when using padding or margin or any other attribute that can be set to a specific side, it is best to use `start` and `end` instead of `left` and `right`. Different languages such as Hebrew and Arabic requires support for RTL direction, and choosing actual sides like left and right isn't the same in English. `start` and `end` will set the correct side based on the device current language preferences.

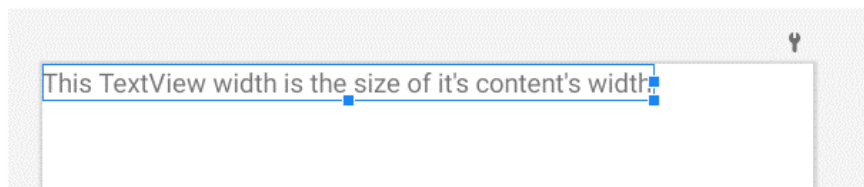
Constants

Instead of exact size in DP units, we can use constants. The most common are the `match_parent` and `wrap_content` constants.

- **match_parent** – instead of defining the view's height and width with specific values, we can set it up to **match the parent's value**. So for example if we set a TextView's width to `match_parent`, it will **hold the whole width of the parent**



- **wrap_content** – instead of defining the view's height and width with specific values, we can set it up to **match its content width**. So for example if we set a TextView's width to `wrap_content`, then it will be the **exact width that is needed to hold the view's content**.



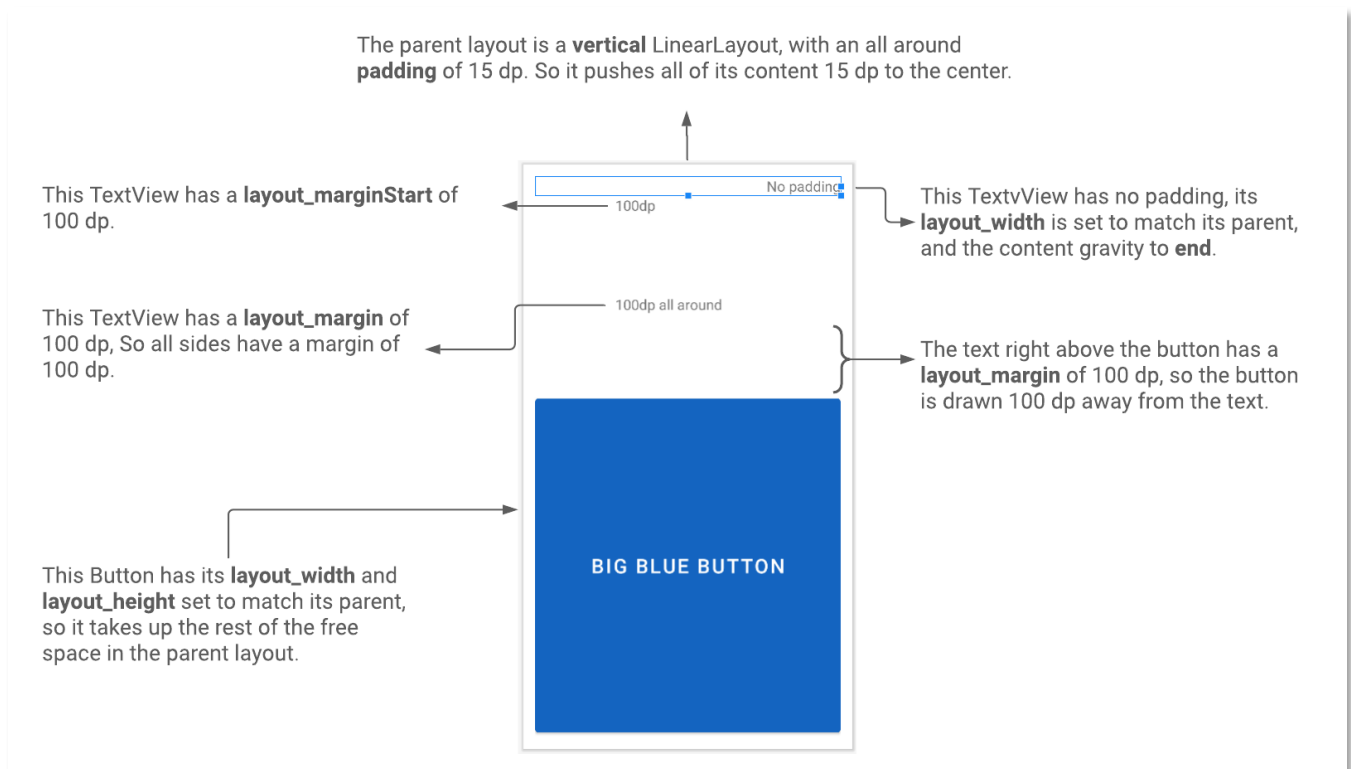
Note: The blue border in the images is not present in the app itself. It is a part of the preview that the Android Studio presents, to show the developer the actual border of the view.

Gravity Constants

We can “anchor” the view to a specific spot in its parent layout on both X and Y axis.

- `android:gravity` sets the gravity of the content inside the view. It is used to control the gravity of all child views in the view it is used on.
- `android:layout_gravity` sets the “outside” gravity of a view. It is used to control the gravity of the view it is used on, inside its parent.

There are a lot more constants that we use when creating a user interface. For example, choosing a vertical `LinearLayout` or a horizontal one is also defined by a constant. Constants are basically presets that we can use in order to define the different views. During this tutorial, it is very recommended to play around with them, and see different results.



XML Files

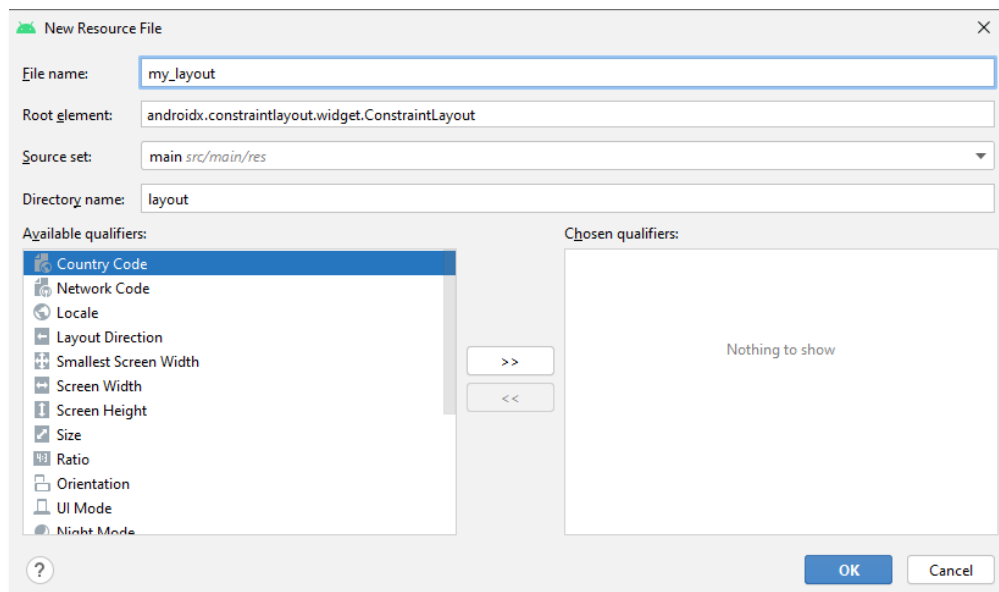
XML stands for Extensible Markup Language. Through these files we define our views and resources. There are more than one type of XML resource file and each one serves a different purpose:

- **Layout XML files** – Contains data about all the views and widgets. These files are inside the `res/layout` folder
- **color.xml** – xml type that holds our predefined colors. This file is inside the `res/values` folder.
- **strings.xml** – xml type that holds our predefined texts. This files is inside the `res/values` folder.
- **themes.xml** – xml type that holds our predefined styles for views. This file is inside the `res/values` folder.
- **drawable** – xml type containing graphics such as backgrounds and effects. This file is inside the `res/drawable` folder, alongside raw images
- **Animation XML files** – xml type that holds animation settings. This file requires us to create a new directory named `anim`, under the `res` folder.
- **manifest.xml** – This file contains essential information about the app such as activities, services used, permissions, app icon, RTL support and more. This file is inside the `app` folder.

For now, we will focus on the Layout XML. When creating a new layout file, we often start with a layout that will be responsible for holding all the other views and widgets in the layout.

Note: Layout resource files may also have a single view in them, without any sub views or a parent layout. Discussed later on.

Inside the res folder of the project, there's a layout package that contains all the layouts of the application. To create a new layout, right click on the layout package and select **New -> Layout Resource File**.



You can choose the Root element when creating the file, or simply change it after that. After clicking OK, a new layout file will open with the selected layout as the Root layout:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <androidx.constraintlayout.widget.ConstraintLayout
3     xmlns:android="http://schemas.android.com/apk/res/android"
4     android:layout_width="match_parent"
5     android:layout_height="match_parent">
6
7 </androidx.constraintlayout.widget.ConstraintLayout>

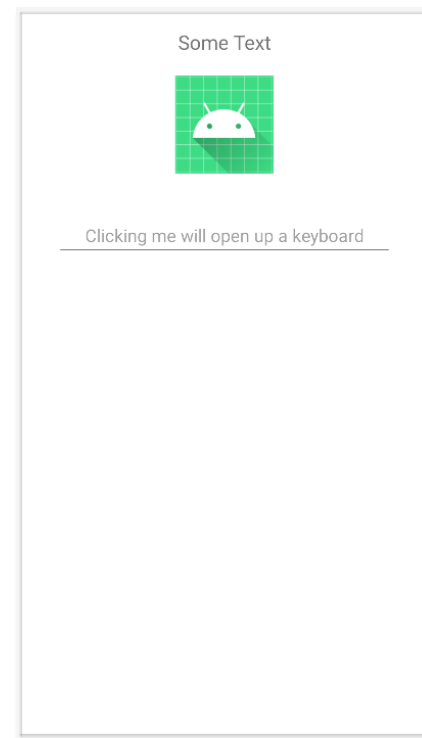
```

Take a look at the following XML code and its preview:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent"
5     android:orientation="vertical"
6     android:padding="15dp">
7
8     <TextView
9         android:layout_width="match_parent"
10        android:layout_height="wrap_content"
11        android:gravity="center"
12        android:text="Some Text"
13        android:textSize="20sp" />
14
15    <ImageView
16        android:layout_width="100dp"
17        android:layout_margin="20dp"
18        android:layout_height="100dp"
19        android:layout_gravity="center"
20        android:src="@mipmap/ic_launcher" />
21
22    <EditText
23        android:layout_width="wrap_content"
24        android:layout_gravity="center"
25        android:layout_margin="20dp"
26        android:layout_height="wrap_content"
27        android:hint="Clicking me will open up a keyboard" />
28
29 </LinearLayout>

```



There are 3 types of views inside a **vertical** LinearLayout: **TextView**, **ImageView** and an **EditText**.

Lines 2- 29: LinearLayout

As mentioned, This is a ViewGroup type, so it can have other views inside it.

The layout's attributes will be inside the starting tag, and if it has views inside it, we need to provide a closing tag (line 29).

```

2 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent"
5     android:orientation="vertical"
6     android:padding="15dp">

```

Attributes:

- **layout_width** and **layout_height** – these attributes specify the basic width and height of the view. These are required for all views. By setting the width and the height, the parent view knows how much space the view is going to hold. In the image above, The main layout is set to match its parent height and width, so it holds all the space in the screen. To be exact, it takes all the space its parent will give it.

- **orientation** – As mentioned before, we can set up views in this layout vertically or horizontally. In this case it's **vertical**, so each view will be presented in the order that we put it in the XML code.
- **padding** – a padding of 15 DP is set on the LinearLayout, so all its children are pushed 15 DP into the center.
- **xmlns** – XML namespace. Used to avoid long repetitive prefixes. So instead of the whole path, we just use **android**

Lines 8- 13: TextView

This widget provides us with space for simple text. It cannot hold other views, so a closing tag is not needed.

```

8      <TextView
9          android:layout_width="match_parent"
10         android:layout_height="wrap_content"
11         android:gravity="center"
12         android:text="Some Text"
13         android:textSize="20sp" />

```

Attributes:

- **layout_width and layout_height** – the TextView is set to match its parent width. Its height is set to wrap its content, so the height of the TextView is as the text height ().
- **gravity** – Because we've set the TextView's width to match its parent, we can center the text to be exactly at the center of the parent layout by centering the text inside the TextView.
- **text** – This attributes lets us define the text of the TextView.
- **textSize** – We use SP units to define the texts size.

Lines 15- 20: ImageView

This widget provides us with space for an image or a drawable. It cannot hold other views, so a closing tag is not needed.

```

15     <ImageView
16         android:layout_width="100dp"
17         android:layout_height="100dp"
18         android:layout_gravity="center"
19         android:layout_margin="20dp"
20         android:src="@mipmap/ic_launcher" />

```

Attributes:

- **src** - The **src** attribute takes a path to an image or a drawable, and displays it within the bounds of the view.

- **layout_margin** – sets the space between its borders and the rest of the layout.
- **width and height** – set to be 100 DP. Actual sizes may varies in different screen densities.

Lines 22- 23: EditText

This widget provides us with space for user input.

```
22 <EditText
23     android:layout_width="wrap_content"
24     android:layout_height="wrap_content"
25     android:layout_gravity="center"
26     android:layout_margin="20dp"
27     android:hint="Clicking me will open up a keyboard" />
```

Attributes:

- **layout_width and layout_height** – the EditText's height and width are set to wrap its content.
- **gravity** – Because we've set the EditText's width to wrap its content, we can't use gravity to try to center it in its parent layout. To do that we need to use layout_gravity. Setting the layout_gravity to center will put the content to be exactly at the center.
- **text** – This attributes lets us define the text of the TextView.
- **textSize** – We use SP units to define the texts size.
- **hint** – sets a text that is shown until an input is inserted into the view.

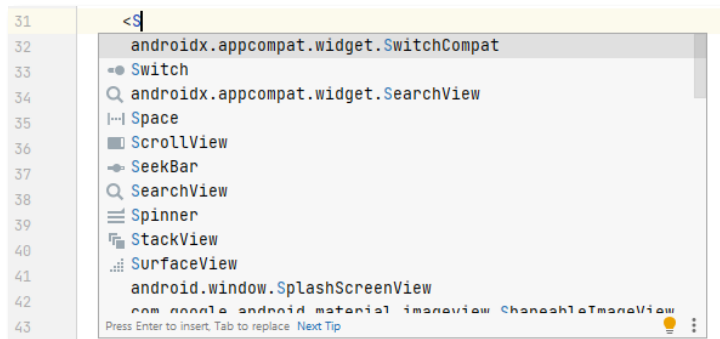
Note: A ViewGroup can contain other views so we close out its starting tag with a right arrow symbol to indicate it isn't "closed", after declaring its attributes. A regular View cannot hold other views, so its starting tag is closed with a right arrow **prefixed with a slash symbol** ("/"), which completely closes out the view, just like a full closing tag closes a layout (line 29).

Note II: all attributes of all views will be overridden once changed in runtime.

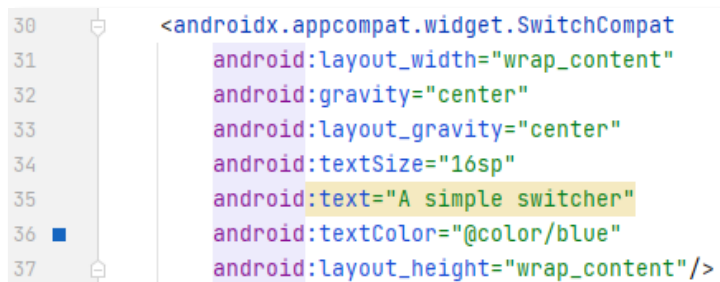
Adding Views

We can add views to our app either by putting them inside an XML file, or by code. As mentioned, all views have a set of attributes that are available to be configured. Setting up attributes in the XML file will make them known at compile time, and the Android Studio will be able to provide a preview for them.

Let's focus on adding views by XML first. In order to add a view by xml, we start with a left arrow symbol (" $<$ ") and then the Name of the view. The Android Studio has a built in sentence completion system, so once you start typing your view, it will offer you a list to choose from.



Once you choose the view you want, it's time to decide its initial attributes.



AppCompatActivity views: Views that have "Compat" in their names are views that support compatible features on older versions. Some views such as a **Button** or an **EditText** are actually using the compatible version automatically.

After we've set the view in the xml file, we need a way to interact with it. In order to do that, we need to uniquely identify it, and then bind to it by code. Any View object can have an ID so it can be uniquely identified. We may repeat the same ID in different layouts and the compiler will recognize which view we are referencing.



In line 31, an ID attribute has been given to the Switch view. We write the ID as a string, but it is translated into an Integer that is associated with the view. Let's break this line down:

- @ - This symbol at the beginning indicates that this is an ID resource.
- + - This symbol means that this is a new resource name that must be added to our resources.
- id/ - after these characters, the actual ID starts.
- **my_switcher** – the actual ID.

Note I: The syntax for giving a view an ID is the same for all views.

Note II: Sometimes not all views require an id, just the ones we need to reference. In some other times, ids are required in order to parse the XML and preview the correct data (for example: `MotionLayout`).

Note III: The `SwitchCompat` view is a `CompoundButton` type which inherits from the `Button` view. This kind of button has two states - checked and unchecked. When the button is clicked, the state automatically changes. `CheckBox` and `RadioButton` views are `CompoundButtons` as well.

Binding to Views

After adding the view to our app, we can launch it and see our new view. But if we need to interact with the view's properties, or add a listener to it, we can't because we don't have any control over it. In order to make it do the things we want, we must bind to it. So after giving a View an ID, we can reference it using the `AppCompatActivity.findViewById()` method.

`AppCompatActivity.findViewById()`

This method allows us to search for an id within all the views of an activity. The `View` class also has this method but will strictly search the view itself and its sub views, and not the whole activity.

The `findViewById()` method is written in Java and returns an instance of a view that is a [Platform type](#) ("!" symbol), meaning referencing it would be as if it isn't nullable, even though it can be. This is in order to treat them the same as in Java in terms of nullability.

The returned instance needs to be specified:

```
val mySwitcher: SwitchCompat = findViewById(R.id.my_switcher)
```

Specify the variable type

And as of Android 8.0:

```
val mySwitcher = findViewById<SwitchCompat>(R.id.my_switcher)
```

Specify the returned View type and infer to variable

Be Careful!

```
val mySwitcher: ImageView = findViewById(R.id.my_switcher)
```

The above line will compile, but also crash the app with the following exception:

```
Caused by: java.lang.ClassCastException: androidx.appcompat.widget.SwitchCompat cannot be cast to android.widget.ImageView
```

This is an easy error to fix, but the point is that this syntax is prone to errors. Even though the type is declared, we might accidentally use the wrong view. It is possible because the findViewById method returns a generic type that extends the View class.

```
@Override
public <T extends View> T findViewById(@IdRes int id) {
    return getDelegate().findViewById(id);
}
```

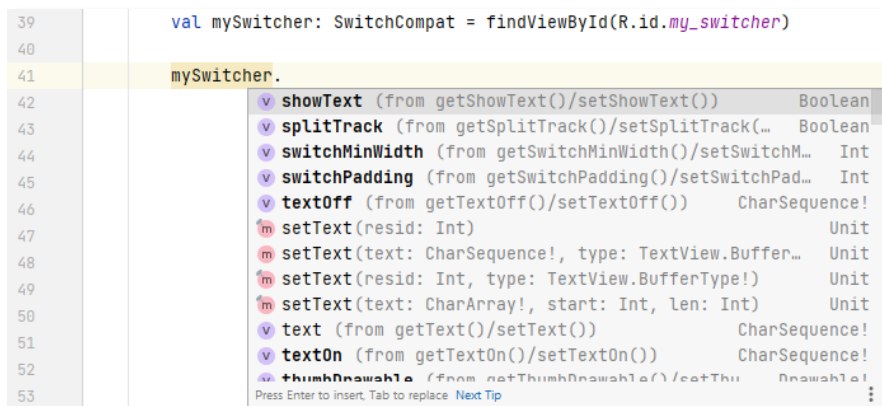
AppCompatActivity – findViewById() source code.

We won't be warned about it during compilation and the app may not crash at launch. You will find out about it when you try to invoke a field or a method that belongs to the view we initially wanted. The app will either crash or have some unexpected behavior. So make sure you are binding to the correct kind of view.

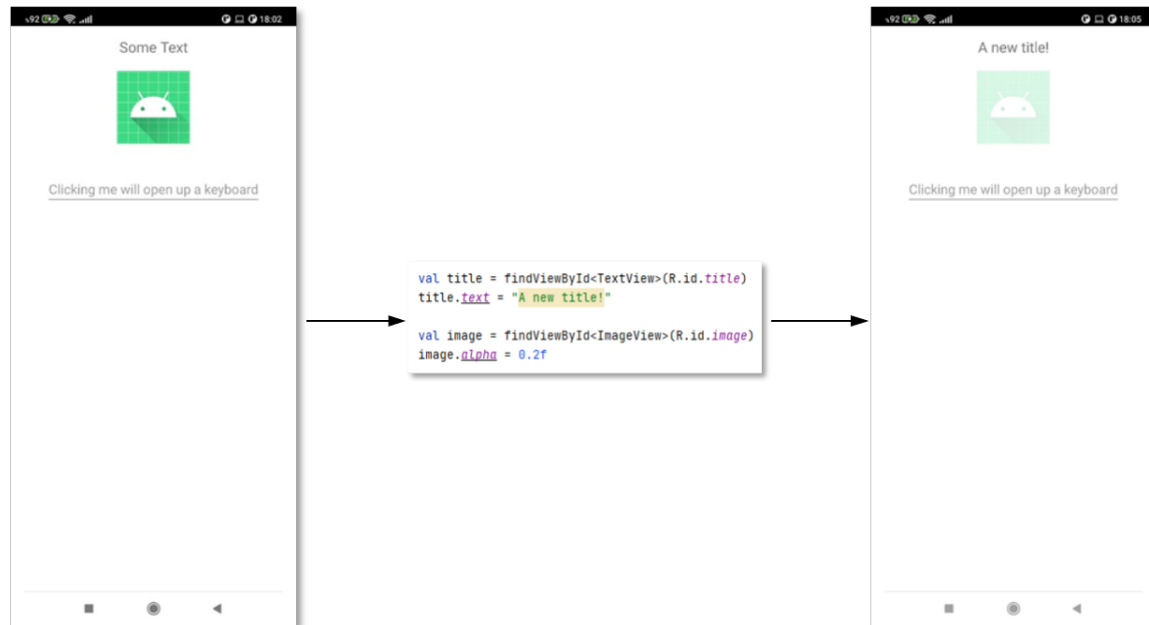
Note: Keeping lots of references for views in the code can be pretty messy and tedious. These days there is a much better approach for keeping track of all the views inside a layout – **ViewBinding**. This approach makes sure you have the proper view type, without explicitly specifying it. In addition, This approach makes sure the reference received isn't nullable. More on that subject later on.

Interacting With Views

After getting a reference to the view, we can access its public fields and methods.



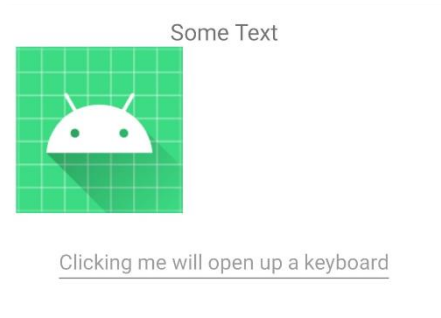
Changing elements on screen is easy as that:



Its recommended to explore the different attributes. Just remember that programmatic changes override any settings you have in the XML files.

Here's another example where the `ImageView` is resized and its `layout_gravity` set to `START`

```
val image = findViewById<ImageView>(R.id.image)
val params = LinearLayout.LayoutParams( width: 400, height: 400)
params.gravity = Gravity.START
image.layoutParams = params
```



In order to resize the image, we need to access it's layout attributes, and provide the parent with the new parameters.

In this example, The `ImageView` is inside a `LinearLayout`, so `LinearLayout.LayoutParams()` was used. There are more subclasses of `ViewGroup` that has their own `LayoutParams` such as `RelativeLayout`, `TableLayout`, `TableRow`, `RadioGroup`, and more. Use the one that corresponds to your parent layout.

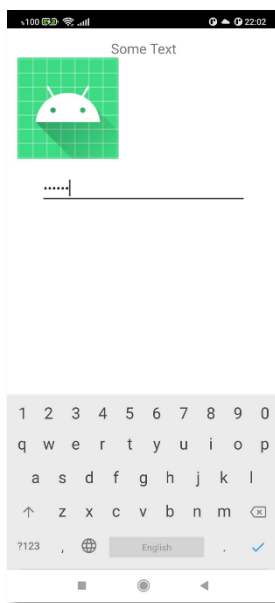
After a view is changed, it needs to be redrawn to reflect the changes. If a property such as a background or text is changed, than only the `invalidate()` method should be called to signal that the layout needs to update itself. But if something that affects the size will change, than `requestLayout()` should be called as well. `requestLayout()` will trigger the

`onMeasure()` and `onLayout()` methods not only for the view itself, but also for its parent - all the way up the view tree.

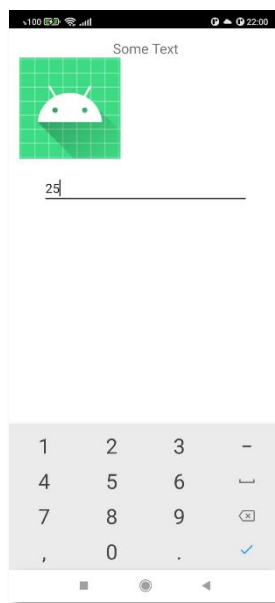
When changing layout parameters, **we don't need to call these functions ourselves**. It is already present in the `setLayoutParams()` function.

Note: The dimension values passed in the `LayoutParams()` are Integers, **not dp**. It's the developer's job to convert it to dp units in order to display the view correctly.

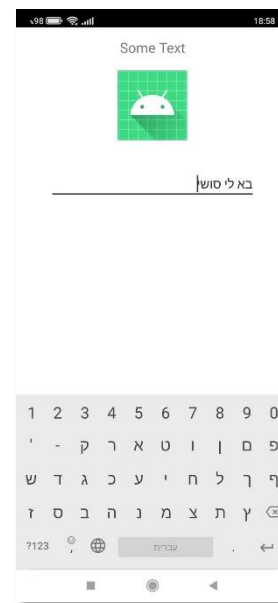
Clicking on the `EditText` will pop up the keyboard. Different keyboards are suited for different tasks.



android:inputType="textPassword"



android:inputType="number"



Default input type

User Touch Interaction

Touch recognition and handling are critical parts of developing user interaction. Handling different types of clicks, drags, swipes etc. are basic ways for the user to interact with the app. An app can be controlled by voice and motion as well, but we will focus on physical touch interactions.

In the core of all UI gestures are the **touch events**. the `View.OnTouchListener` interface lets us register to a view's touch events and intercept them. This interface has one abstract method – `onTouch`. The `onTouch` method has access to the `MotionEvent` data, which describes the type of touch action that happened by using constants, and coordinates on the screen.

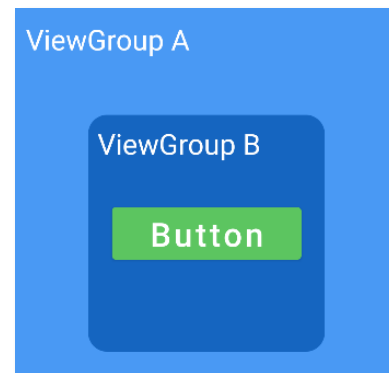
A simple gesture is composed of the following ACTIONS:

1. **DOWN** – The user pressed down on the screen.
2. **MOVE** – The user moved around during a press gesture (between ACTION_UP and ACTION_DOWN).
3. **UP** – (or CANCEL) A press gesture has finished. The user released the screen.

The full collection of constants can be found [here](#).

The touch event is first passed to the Activity. It is then passed to “**ViewGroup A**” `onInterceptTouchEvent()`, to let it have a chance to consume the event by returning true.

If the event wasn’t consumed, its passed to “**ViewGroup B**”, which also has an opportunity to intercept and consume the event. It is then passed to the Button (the actual view that was pressed) and its `onTouchEvent()` is triggered.



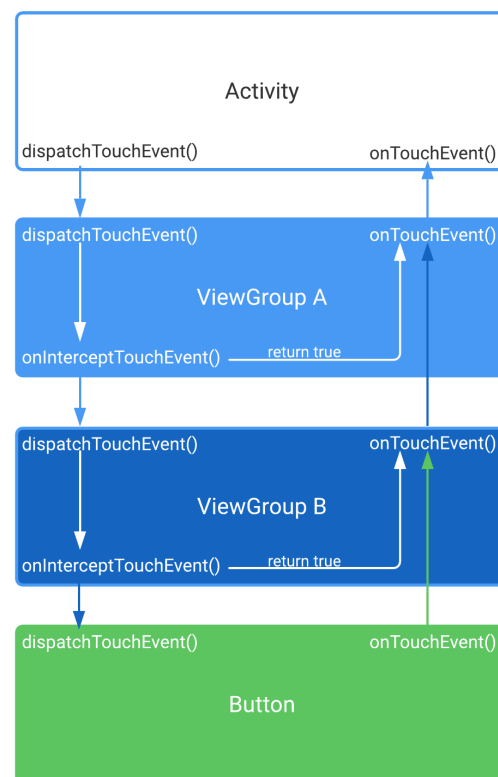
Every touch event may be propagated through the view tree. Every view that contains the touched view, will have an opportunity to respond to the touch event.

The first to be notified about the touch event is the Activity. After that, The different view groups are notified, going all the way to the specific view that was touched.

After everyone is notified, everyone is given a chance to handle the event.

The first one to handle the event (if not intercepted), is the specific view that was touched – the view on top. The last one to have a chance to handle it is the activity. So the activity is the first to know about the touch event, and the last to handle it.

If some ViewGroup needs to handle the touch event immediately and not give anyone else a chance to handle it, then we can simply return true in its `onInterceptTouchEvent()` method (or for the activity, override `dispatchTouchEvent()`).



Note: The **dispatcher** is responsible for identifying which methods to invoke next. The dispatcher triggers the `onInterceptTouchEvent()` of a ViewGroup, and then it triggers the `dispatchTouchEvent()`

Event Handling

In order to respond to events such as clicks, swipe gestures, text inputs etc. we need a way to know about them. The approach is to capture the events from the specific view that was interacted with, by implementing an interface that registers for the view's click event. This way we can intercept it and add our own logic.

The View class provides the tools to do that. The View class contains a collection of interfaces that allow us to define our callback. A **callback** is a function that will execute once some other function has finished. So for example, if we want to respond to a button click, we need to provide an implementation of the [OnClickListener](#) interface to the [View.setOnClickListener](#) method. The following code shows an **Anonymous implementation** of the `OnClickListener` interface:

```
val button = findViewById<Button>(R.id.my_button)

button.setOnClickListener(object : View.OnClickListener {
    override fun onClick(p0: View?) {
        TODO(reason: "Not yet implemented")
    }
})
```

You can also define a class to implement the `OnClickListener` interface. In the code below, the MainActivity class is responsible for implementing the `onClick` method. Notice `"this"` has been passed as the listener for the button.

```
31 class MainActivity : AppCompatActivity(), View.OnClickListener {
32
33     override fun onClick(p0: View?) {
34         TODO(reason: "Not yet implemented")
35     }
36
37     override fun onCreate(savedInstanceState: Bundle?) {
38         super.onCreate(savedInstanceState)
39         setContentView(R.layout.drill_layout2)
40
41         val button = findViewById<Button>(R.id.my_button)
42         button.setOnClickListener(this)
```

The `OnClickListener` interface contains one abstract method – `onClick(v: View)`. Because there's a **Single Abstract Method** in that interface, we can use a [SAM interface](#). Instead of implementing the interface manually, we can use a lambda expressions that matches the signature of the interface implementation:

```
val button = findViewById<Button>(R.id.my_button)
button.setOnClickListener { it: View!
    TODO(reason: "Not yet implemented")
}
```

Note: a **lambda expression** is a block of code that can be passed as a parameter to other functions. These type of functions are called Higher-order functions.

Inside the callback, we are provided with the view that was clicked, and can access it by using the `it` keyword. For example, this code will change the color of the button when it's clicked:

```

val button = findViewById<Button>(R.id.my_btn)
button.setOnClickListener { it: View!

    val color: Int = Color.rgb(
        (0..256).random(),
        (0..256).random(),
        (0..256).random()
    )

    it.setBackgroundColor(color)
}

```

There are a lot of different Listeners. Some are shared by all views, like the `View.OnClickListener` interface, and some are designed for different types of views to handle inputs and events.

```

val title = findViewById<TextView>(R.id.title)
val editText = findViewById<EditText>(R.id.edit_text)
editText.addTextChangedListener(object : TextWatcher {

    override fun onTextChanged(text: CharSequence?, p1: Int, p2: Int, p3: Int) {
        title.text = text
    }

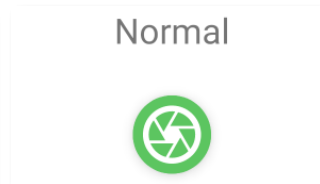
    override fun beforeTextChanged(p0: CharSequence?, p1: Int, p2: Int, p3: Int) {}
    override fun afterTextChanged(p0: Editable?) {}
})

```

```

btn.addOnShrinkAnimationListener(object: AnimatorListenerAdapter() {
    override fun onAnimationEnd(animation: Animator?, isReverse: Boolean) {
        super.onAnimationEnd(animation, isReverse)
        title.text = "Normal"
    }
})

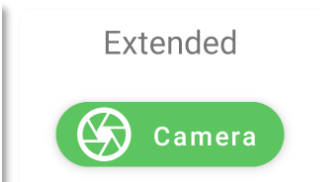
```



```

btn.addOnExtendAnimationListener(object: AnimatorListenerAdapter() {
    override fun onAnimationEnd(animation: Animator?, isReverse: Boolean) {
        super.onAnimationEnd(animation, isReverse)
        title.text = "Extended"
    }
})

```



Resources

Besides the code of the app, there are additional files that hold different resources like images, colors, strings, animations, navigation, and the layouts we've seen. These resources are kept and maintained in separate directories, under the **res** directory. This way nothing is hardcoded into the code, and we can provide alternative resources for specific devices and configurations.

We'll start by going over some common resources.

Drawable resources

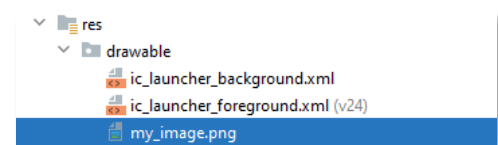
A drawable resource is a general concept for a graphic that can be drawn to the screen. We can create new drawable resources by adding them into the drawable package inside the res folder of the project. We can then load these files using the view's attributes. For example, The ImageView and ImageButton views have a `src` attributes which can take a drawable as a value and display it.

We can also use drawables as backgrounds and create different shapes with different behaviors in order to change the appearance and feel of a view.

There are several different types of drawable, some of the most commons are:

Bitmap File

We can load images (.jpg, .png, .gif) into views that support them such as an ImageView, ImageButton and more. To add an image to your project, simply copy it into the drawable package inside the res folder of the project.



XML File

To create, right click on the drawable package and select New -> Drawable Resource File. This will open up a dialog, asking for a name for the file.

The XML drawable allows us to define shapes with specific styles and then apply them on a view. Furthermore, we can combine drawables into a [StateListDrawable](#) - a type that references different bitmap files for different states. The root element of this type of drawable is the `selector`, where each state drawable is defined in a nested `<item>` element.



StateListDrawable XML file example

Another use case is defining a shape that will act as a background for a view:



rounded_bg.xml

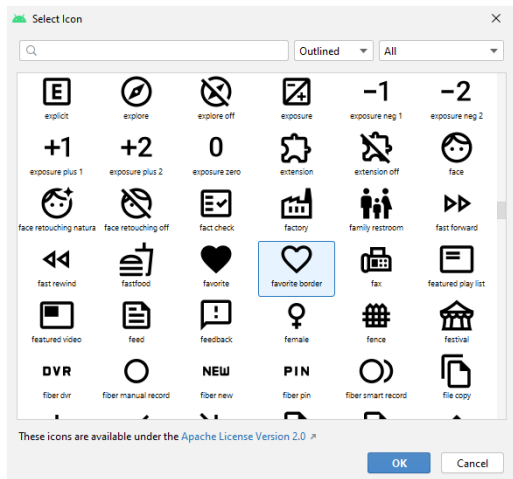


Button view with "round_bg" as background



Vector drawable

a graphic defined in an XML. This type of drawable has a major advantage – image scalability. Meaning, the same file is resized for different screen densities without loss of quality. To create a Vector drawable, Right click on the drawable package, choose New -> Vector Asset. You can load .SVG files or choose a Clip Art:



```
<com.google.android.material.button.MaterialButton
    style="@style/Widget.MaterialComponents.Button.OutlinedButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/like"
    android:textAllCaps="false"
    android:textColor="@color/pink"
    android:textSize="16sp"
    app:cornerRadius="10dp"
    app:icon="@drawable/icon_favorite"
    app:iconTint="@color/pink" />
```

Optional research: The `MaterialButton` is a part of [Material Design](#), which is highly practiced. It inherits from `AppCompundButton` and has more attributes such as `cornerRadius` (previously this attribute was available through a drawable).

During the course, It is recommended to get familiar with Material Design and work with its different Views and ViewGroups.

Strings Resource

One of the most common resource is the string resources, that enables us display different languages based on the device configuration. You can access them by using the IDs you gave them. A String resource provides text strings for the application with optional formatting:

- String – A single string. Can be referenced from the application with parameter, or from other resource files.
- String Array – A single array of strings.
- Quantity Strings – Different languages have different grammar for quantity. Android provides methods to select the appropriate strings.

This type of resource is managed and stored in a way that lets the Android system display the text in the correct language according to the device's current configurations.

To add a new string resource, simply create a new entry in the `strings.xml` file, inside the `<resource>` tag:

```

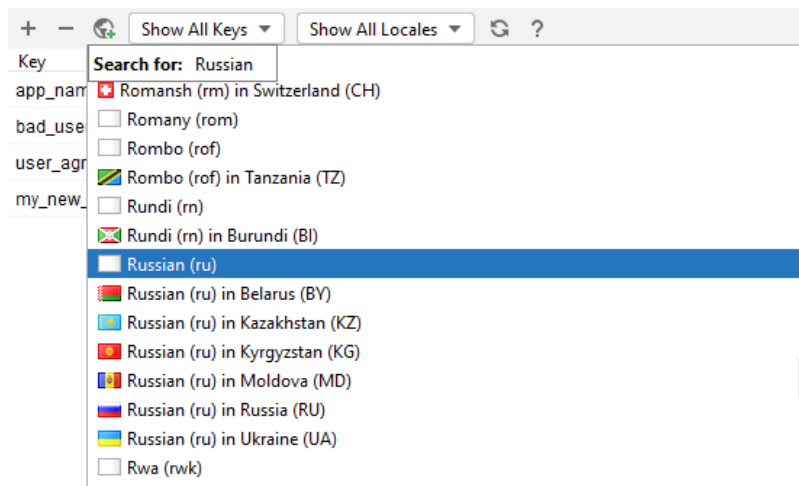
1 <resources>
2   <string name="app_name">DrillApp</string>
3
4   <string name="bad_username">Your user name does not meet the requirements. Please select a new one.</string>
5
6   <string name="user_agreement">
7     Lorem ipsum dolor sit amet, consectetur adipiscing elit.
8     Mauris commodo mauris ac libero rutrum.
9     Ut rhoncus auctor lectus ac sodales. Curabitur commodo accumsan tincidunt.
10    Pellentesque varius ornare nunc, porttitor egestas quam pretium ac.
11  </string>
12
13  <string name="my_new_string">My new String</string>
14
15 </resources>

```

The `strings.xml` file is created automatically when you start a project, but if you need to provide translated strings, then you need to create a new values directory, and give it the proper name for the translation.

This can be achieved by switching to Project view in the Project tab, and creating a new package named `values-{language code}`. For example, `values-iw` would be the package name used for Hebrew strings. The name of the resource needs to be identical to the default name and the `iw` suffix is one of the language codes that can be provided as a language qualifier (explained later on).

We don't need to do it manually, Android Studio provides an editor for strings. In the XML file click on [Open editor](#) at the top right side. Through there we can add more strings, and provide additional translations.



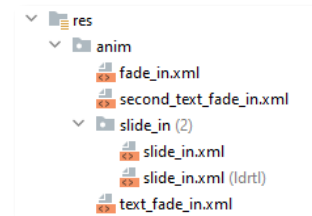
The world icon shows a list of all supported languages

Animation resources

Animations are used to give the UI a rich look and feel. Animations in android apps can be performed through XML or Kotlin code. Either way, we are using properties to transform a view from one visual state to another.

In the Drawable section of the tutorial, we saw how we can display different drawables for a specific state. This is not the same thing. Animations are based on frames where the properties change when the frame changes, and we only define the starting stage and the completion stag. We can also define the time of the animation, and make the animation loop itself indefinitely.

To apply Animations to our application via XML, we need to make a folder called **anim** under the **res** folder to store animation files of the application.



The **slide_in** folder inside the anim folder contains files with the same name, but Android will load the correct one based on the device current direction: Right-to-Left or Left-to-Right. This is part of Resource Localization, which is explained later on.

Below are some XML animations examples:

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <set xmlns:android="http://schemas.android.com/apk/res/android">
3   <alpha android:fromAlpha="0.1"
4         android:duration="3000"
5         android:toAlpha="1"/>
6 </set>

```

Fade in animation. By using the alpha property, we can create a fade in animation. Start value is 0.1 and end value is 1. We can make it a **fade out** animation simply by converting the values.

```

1 <set xmlns:android="http://schemas.android.com/apk/res/android">
2   <translate
3     android:duration="700"
4     android:fromXDelta="0%"
5     android:fromYDelta="0%"
6     android:toXDelta="100%"
7     android:toYDelta="0%" />
8 </set>

```

Slide in animation. By using the translate tag, we can animate a view from one place on the screen to another. Left and right animations are done on the X axis. Up and down animations are done on the Y axis.

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <set xmlns:android="http://schemas.android.com/apk/res/android">
3   <rotate
4     android:fromDegrees="0"
5     android:toDegrees="360"
6     android:pivotX="50%"
7     android:pivotY="50%"
8     android:repeatCount="infinite"
9     android:duration="1200" />
10 </set>

```

Rotate animation. This animation file will make the view rotate on its pivot (x and y coordinates of the pivot are 50% so the center of rotation will be the center of the view). The animation will repeat itself indefinitely because **repeatCount** is using the "infinite" constant.

After creating the animation resource, we need to load it inside our code, and apply it on the target view:

```
50     val slideInAnim = AnimationUtils.loadAnimation(  
51         applicationContext,  
52         R.anim.slide_in  
53     )  
54  
55     val btn = findViewById<MaterialButton>(R.id.order_btn)  
56     btn.startAnimation(slideInAnim)
```

To apply Animations directly to a view via code, we can use an [ObjectAnimator](#).

ObjectAnimator is a Subclass of [ValueAnimator](#), which allows us to set a target object and apply animations to it by changing the actual properties of the object:

```
val fadeIn = ObjectAnimator.ofFloat(btn, propertyName: "alpha", ...values: 0f,1f)  
fadeIn.duration = 4000  
fadeIn.start()
```

There are more types of animations and each one has properties and parameters that can be used to tweak and optimize the animation process. It is recommended to experience with the different animation libraries.

We can also set listeners for the animations and implement logic for when it starts and ends. By doing so we can create a **chain of animation** that will play animations one after another. Android also provides functions that lets us define more easily the way animations are played.

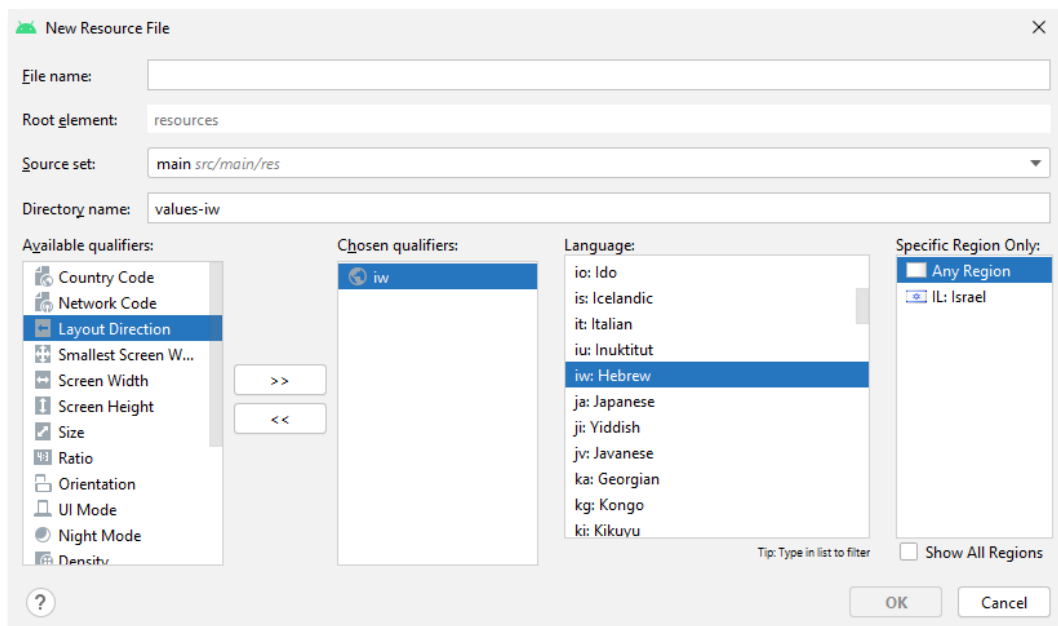
```
val btn = findViewById<MaterialButton>(R.id.order_btn)  
  
val translationAnimator: Animator =  
    ObjectAnimator.ofFloat(btn, propertyName: "translationY", ...values: 0f, -100f).setDuration(1000)  
  
val alphaAnimator: Animator =  
    ObjectAnimator.ofFloat(btn, propertyName: "alpha", ...values: 1f, 0f).setDuration(1000)  
  
val animatorSet = AnimatorSet()  
  
// Play animations one after another  
animatorSet.playSequentially(translationAnimator,alphaAnimator)  
  
// Play animations at the same time.  
animatorSet.playTogether(translationAnimator,alphaAnimator)  
  
animatorSet.start()
```

The code above is using `Animators` and an `AnimatorSet`, to play animations either sequentially or all at once.

It's the small details the make a user experience – a great one!

Resource Localization

Android studio lets us define different qualifiers when creating a resource. For example, we can create a new string.xml using the Locale qualifier:

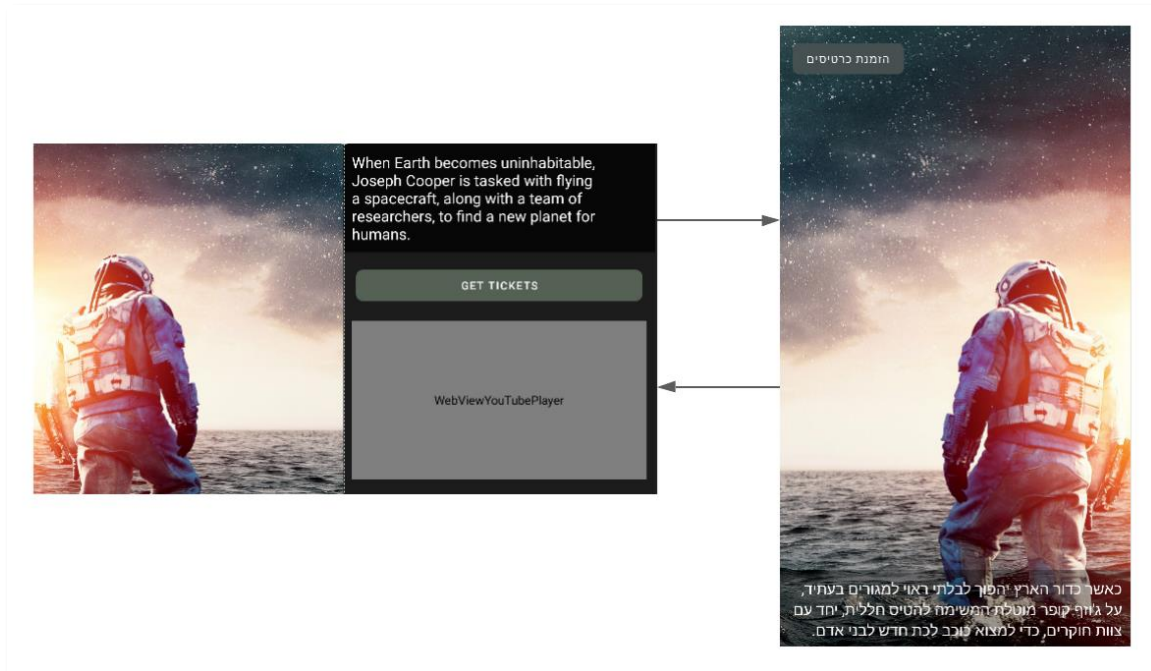


Note: You can add multiple qualifiers to one directory name, by separating each qualifier with a dash. If you use multiple qualifiers for a resource directory, you must add them to the directory name in the order they are listed [HERE](#).

In general, qualifiers define the type of resource to use at runtime. Android detects the current device configuration and loads the appropriate resource. When creating a new resource we are given a list of qualifiers to choose and configure. They are all self-explanatory, and easy to use.

So instead of opening a new package with the proper suffix, we can use a qualifier, and Android Studio will generate the appropriate files.

The example below shows one app that loads different drawables, views and view groups based on the device configuration.



In the example above, Android uses the appropriate resources based on location, language, screen size, orientation and more.

If you choose to let the app have a [Landscape](#) mode, You might need to provide a different layout. Your Kotlin code will be exactly the same, but Android will load a different XML layout resource, based on the best match for the device configurations.

Dialogs

There are many times in the application that we need to ask the user for a decision or show critical information. A highly practiced way to do that is by using dialogs. A dialog is a window that appears in front of the app's content.

We need the user's full attention, so dialogs disable all app functionality when they appear, and remain on the screen until confirmed, dismissed, or a required action has been taken. Dialogs are intentionally interruptive and should be used to display critical information that requires the user to acknowledge it, and sometimes provide an input.

The [Dialog](#) class is the base class for dialogs, and it has subclasses with pre-defined UI and functionality.

AlertDialog

This type of dialog has a pre-defined UI that can show a title, up to three buttons and a list of selectable items. We can also use custom layout, instead of the pre-defined layout. To create an AlertDialog we need to use its Builder class:

`AlertDialog.Builder`.

We can't create an AlertDialog directly, because its constructors are protected. Instead, we are given a builder to configure the content to show, and callbacks to user interactions.

```
val builder: AlertDialog.Builder = AlertDialog.Builder(context: this)
```

The builder has public methods to configure the dialog:

```
val builder: AlertDialog.Builder = AlertDialog.Builder(context: this)

// Build the dialog using its Builder class
builder.apply { this: AlertDialog.Builder
    setTitle(R.string.dialog_title)
    setMessage(R.string.dialog_message)
    setCancelable(false)
    setIcon(R.drawable.icon_exit)

    setPositiveButton(R.string.yes) { p0, p1 ->
        finish()
    }

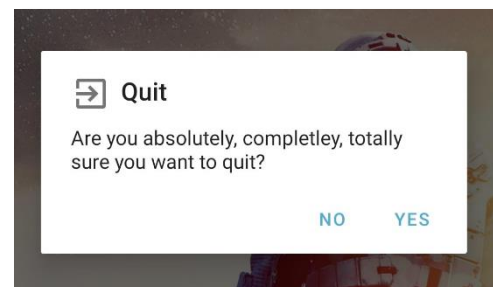
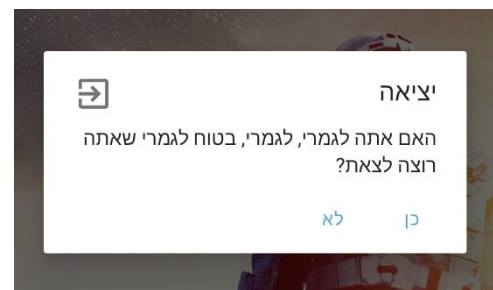
    setNegativeButton(R.string.no) { p0, p1 ->
        // do nothing
    }
}

// Create the dialog
val dialog = builder.create()

// Display the dialog
dialog.show()

// Call dismiss to dismiss the dialog
dialog.dismiss()
```

A simple dialog that cannot be canceled, unless the user clicks on a button.



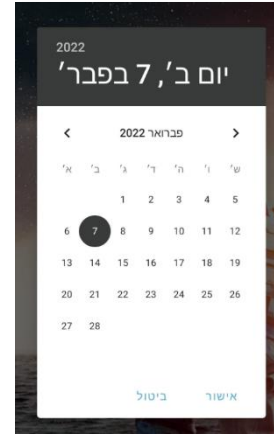
The `create()` method returns an AlertDialog instance that has a `show()` method.

There's also the **DatePickerDialogs** – a simple dialog containing a DatePicker:

```
val c = Calendar.getInstance();
val listener = DatePickerDialog.OnDateSetListener { p0, p1, p2, p3 ->
    // Do something with the input
}

val datePickerDialog = DatePickerDialog(
    context: this,
    listener,
    c.get(Calendar.YEAR),
    c.get(Calendar.MONTH),
    c.get(Calendar.DAY_OF_MONTH)
)

datePickerDialog.show()
```



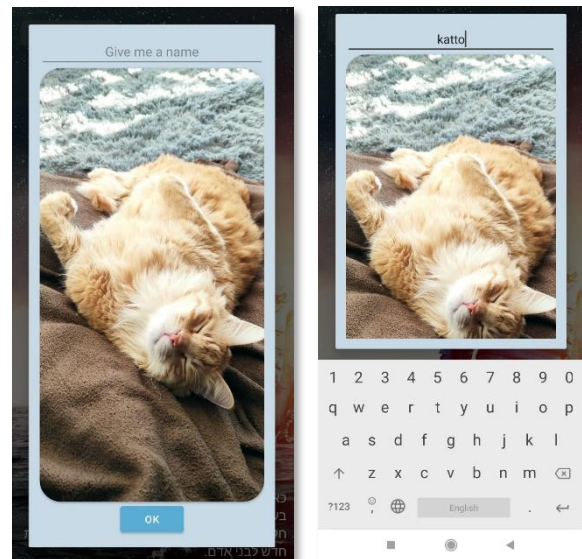
We can also make our own layout and set it to be the layout of the dialog:

```
val dialogBuilder = AlertDialog.Builder(context: this)
val dialogView: View = LayoutInflater.inflate(R.layout.dialog_layout, root: null)
dialogBuilder.setView(R.layout.dialog_layout)

dialogView.findViewById<EditText>(R.id.input).setOnClickListener { it: View!
    // Do something with input
}

val alertDialog = dialogBuilder.create()
alertDialog.show()
```

It is common to create a custom dialog, especially if it has more complex logic and views. In that case you should use a **DialogFragment** (will be discussed later in the course).



Android building blocks - Part 1

Activities, Intents, Permissions, Lifecycle and Persistent Storage

Download the full app created in this guide:

https://drive.google.com/file/d/1leLw1Fo7aZfvgL-r2fjwEA_frEfvXS_h/view?usp=sharing

Each Android App consist of four components:

1. **Activity** - this is the main android component. Each Activity represents a full screen. Nowadays we use the `AndroidX AppCompatActivity` which includes more advanced features over the basic Activity which it inherits from. If we go deeper into the inheritance tree you will see that Activity also inherits from Context. The Context is an abstract class whose implementation is provided by the Android system when it is creating the Activity. Yes, the system creates the Activity and not us. The context allows us to interact with the system, it allows access to application level resources and classes. We will need it for each Android API class generation including creating other activities and other app components.
2. **Service** - Service is an app component designed to perform non-UI related operations. Like the Activity the Service also inherits from Context and it allows it to fully interact with the Android system and so massive operations if needed. The fact that the Service doesn't have UI is sometimes an advantage - think about playing music from the background when the user is out of our app UI and wants the music to continue even when he navigates to other apps. An Android App has two states - foreground and idle. Once the app has a foreground activity or a foreground service it is considered to be foreground and has unlimited working power. After they move to the background the app will move to idle state and all of its other service and background activities will be killed by the system.
3. **Broadcast Receivers** - Broadcasts are transmissions that the Android system can send when events occur. For example when connecting an external device, when connecting to a Bluetooth or to Wi-Fi, when a call or SMS arrives or is sent, when the battery changes, when the boot completes, when connecting to a power source, when turning on airplane mode or even when turning the screen on and off. In short for every external Android OS event sends a special broadcast. The Broadcast Receiver is a component that we can register to receive these

broadcasts and do something with them. For example, when we detect that the boot completed we can start a service which let the user know the current weather or checks messages on our server or much more. The Broadcast Receiver doesn't inherit from the Context but receive a limited one by the system when the broadcast it was registered to is sent. This limited context doesn't allow the receiver to do any long term operations but instead it will be finished by the system after 5 seconds if it didn't finish before.

4. **Content Provider** - A Content provider is an app component designed to provide data to other apps. All of the internal system database are arranged in a Sqlite databases and through the Content Resolver the system provides an interface for us to access their stored data like the address book, the content of the SD card, the calendar and other data stored locally on the phone.

Launching the app - order of events:

1. Everything starts in the **Android Manifest xml** file. In that file we set the app name, icon and other initial settings, the required device features and needed permissions but the most important thing, we declare the existence of all of our apps components and their capabilities. When the user installs our app the Android system creates a single instance from each component defined in the Manifest in the JVM Class Loader and uses this instance when it needs to create the component at runtime. It can do so either when the component is asked for explicitly by its name or when his declared capability is needed. This is the situation when the app launches: When the system detects a press on our app icon it sends the MAIN action to our package - the Activity which is registered to that action will be automatically created by the system. Please notice the exported attribute set to "true". This means that this component can be created by the system when its registered action occur.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="il.co.syntax.androidbuildingblocks">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="AndroidBuildingBlocks"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.AndroidBuildingBlocks">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

2. When The Android OS creates an instance of the designated Activity it Automatically calls its **onCreate()** Lifecycle event function. By overriding this function we get our first entry point to the activity creation. Please note that after the `super()` call you can see the **setContentview()** function - this function receives the id of the initial xml layout file and inflate (inflation is creating objects from the static xml list) all of its views and subviews and populate them on the screen.

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

3. After the initial layout has been created we can do additional

customizations like attaching listeners to buttons, play background music reading data from internal or external storage and populate a list with it and much more.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val btn = findViewById<Button>(R.id.dial_button)
    btn.setOnClickListener { it: View!

        val number = findViewById<TextInputLayout>(R.id.phoneNumberInputLayout).editText?.text.toString()

        Toast.makeText(context: this, number, Toast.LENGTH_SHORT).show()
    }
}
```

But since we use view binding we add this line to the app Gradle

```
buildFeatures {
    viewBinding true
}
```

And our **onCreate** will look like this:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding = ActivityMainBinding.inflate(layoutInflater);
    setContentView(binding.root)

    binding.loginBtn.setOnClickListener { it: View!
        val name = binding.nameInputLayout.editText?.text.toString()
        Toast.makeText(context: this, name, Toast.LENGTH_SHORT).show()
    }
}
```

Intents

In the Android OS Intents is all we have :)

Intent are the way to interact with components, they can create them, pass information to them and more. Note the activities usually doesn't have constructors overloading, this is because we don't use constructors calls to create them but rather pass an Intent to the system and let her create the implement the Context for them. Our first entry point to their creation is the

lifecycle event function **onCreate**. When the system receives an Intent she is reading our intentions from it. Our intentions can be either **Explicit** where we are mentioning our desired Component by it's name or **Implicit** where we mention our desired Action string and the system finds the component for us according to what they declare - usually in the Manifest file. If more the one Component can answer that Action the system lets the user pick one and define it as default. The Implicit launch is the case in the app launch - the MainActivity declare himself to answer the action MAIN in the manifest. When the user installs the app the system creates the activity in the class loader it maps it to that action. When later the user press the icon, the system sends an Implicit intent with Main action and because he can answer it and it creates him.

Explicit Intent

So If we want to use the Explicit intent and start our own LoginActivity we first must create it. We have the short way: File->New->Activity->Empty Activity and give the Kotlin and the XML files a name and that's it. by doing this Android studio does allot for us: First it creates a new Kotlin class that extend AppCompatActivity and override the onCreate, then it creates a template xml file and inflate it in the previously overridden onCreate function. It also add the Activity to the Manifest XML file. So basically it is quite nice.

Please note that the default value this activity has for the **exported** attribute in the Manifest file is false, meaning this activity can be created only explicitly by mentioning of his name, He doesn't have any <intent filter> and won't be initiated by an ACTION like the MainActivity.

```
<activity
    android:name=".LoginActivity"
    android:exported="false" />
```

So to initiate it explicitly! create an Intent and use the context's **startActivity()** function:

```
val intent = Intent( packageContext: this, LoginActivity::class.java )
startActivity(intent)
```

Passing Data

what about the name and other information it needs?

Since we don't have a constructor we use the Intent to pass data upon creation. Each Intent contains a Bundle in his extra field. A **Bundle** is basically an HashMap where the key is a String and the value can be String, Int, Double, Float, array of them and any object that implements either the Java's

Serializable interface or its Android Parcelable implementation. We add this extras to the intent using it's **putExtra()** function and when the system creates the new activity it saves this Intent as his property and we use and the **getStringExtra()** with the same key:

In the calling Activity:

```
val intent = Intent( packageContext: this, LoginActivity::class.java).apply {
    putExtra("user_name",name)
}
startActivity(intent)
```

And in the newly created Activity:

```
class LoginActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityLoginBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.textView.text = intent.getStringExtra( name: "user_name")
    }
}
```

And that's it.

Tasks & Back Stack

Activities in the system are managed as an *activity stack*. When a new activity is started, it is placed on the top of the stack and becomes the running activity -- the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits.

Your Activity is placed on top of your Task. By pressing the back button you kill this activity (like calling finish() from within the activity) and pops it from your back stack. By pressing the home button you take all of your Activity's Task and put them all in the background (after a while the OS will kill them if you won't return to them) and you can bring the task to the front along with all go the activities in it.

One thing you must understand regarding the Activity task is that each intent is creating a new activity instance. If, for example, from activity A you open B and then A again a new instance of activity A will be created. If you want to bring and existing Activity instance forward you need to change the Activity's **launchMode** attribute in this the Manifest file from standard to: **singleTop** - meaning if the activity already present in the top of the stack (it is in the front) it won't be recreated, **singleTask** - the system creates a new task just for the

activity but if an there is already an instance of that activity somewhere in that task the system bring him forward and routes the intent to it. Because it is not created, the already existing instance still holds the old intent in its Intent property. If you want to update this field with the new Intent, you need to override the **onNewIntent(intent: Intent)** function. And the last one **singleInstance** which is the same as before except that the system doesn't launch any other activities into the new task created for that activity (in case we wasn't already present).

For further reading and some nice drawings:

<https://developer.android.com/guide/components/activities/tasks-and-back-stack>

Implicit Intent

Now let's say we want to open an address on a Map, send an email, dial a number, open a browser to a specific site, take a photo, record a video or any other action which we want to preform but **don't really care who will perform it**. For this we have the Implicit Intent. In the Implicit Intent we set the Action String(it can be either one of the system fixed actions or our own custom one if you want - not common) and according to all of the installed d components and their declared abilities - declared with **<intent-filter>** - will be populated for the user to choose from. Please note that when using the Action string we usually set the extra data with the **setData(uri)** function. This functions accepts a URI that corresponds with the Action. For example when using ACTION_DIAL or ACTION_CALL the data is a phone number URI (starts with tel:):

```
binding.dialButton.setOnClickListener { it: View!  
    val number = binding.phoneNumberInputLayout.editText?.text.toString()  
    Toast.makeText(context: this, number, Toast.LENGTH_SHORT).show()  
  
    val intent = Intent(Intent.ACTION_DIAL).apply { this: Intent  
        | data = Uri.parse(uriString: "tel:$number")  
    }  
    startActivity(intent)  
}
```

Run the code. A Dialer with the phone number appears. Nice.

Try changing the ACTION_DIAL to ACTION_CALL. What happens?

Yes, the app crashed! this is because the later action try to actually preform the call while the former just showed a dialer and allowed the user to initiate the call (the first time the system dialer ran it also asked for the permission).

Before moving forward to the permissions please read here a a list of common intent action and their corresponding intent filters - IMPORTANT

<https://developer.android.com/guide/components/intents-common>

Permissions

Runtime vs install time permissions

First we must understand that before android 6.0 (marshmallow - api version 23) all permission were install time, meaning that all we had to do is to add the required permission to the Manifest file like this:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="il.co.syntax.androidbuildingblocks">

    <uses-permission android:name="android.permission.CALL_PHONE"/>

    <application
        android:allowBackup="true"
```

Toady we still need to do it, but for some permissions this is not enough.

In the old way, when the user installed the app he was given two options either to install the app and accept all the permissions without the ability to accept one and deny the other, to revoke them at a later time or even to understand exactly when they are using them or simply not installing the app.

Today for some permissions this is still the case, these permissions are mostly what I call background use permission (like getting boot and bluetooth or wifi connections that happen usually when we don't have UI present), and what Android defines as not **dangerous** - but **normal** permission. you can find the full list of them here: <https://stackoverflow.com/a/36937109/2826409> (normal means you should only declare in Manifest file even after Android 6 and dangerous is what we are going to discuss here).

But for the most common permissions like calling, location, recording, reading contacts, and more. We must switch to the Runtime permission mechanism and beside writing them in the Manifest like before we must also present a Pop up window at runtime and specifically ask for them, just like in the iOS model - meaning we have to specifically ask for them when we need them and the user must grant us each requested permission. He can later revoke his approval and he can allow one while denying the other. A good practice is to ask for the permission only when we need it.

And this is how we do it:

First let's move the call execution to a separate function.

```
private fun call() {
    val number = binding.phoneNumberInputLayout.editText?.text.toString()
    val intent = Intent(Intent.ACTION_CALL).apply { this: Intent
        data = Uri.parse(uriString: "tel:$number")
    }
    startActivity(intent)
}
```

Now as a part of the new Launcher API that will be discussed later on we need to create the Permission request Launcher with the basic RequestPermission or RequestMultiplePermission Contract and provide a callback which upon approval will initiate the call:

```
class MainActivity : AppCompatActivity() {

    lateinit var binding : ActivityMainBinding

    val callPermissionLauncher : ActivityResultLauncher<String> =
        registerForActivityResult(ActivityResultContracts.RequestPermission(), ActivityResultCallback { it: Boolean!
            if(it)
                call()
            else
                Toast.makeText(context: this, text: "can't call without permission", Toast.LENGTH_SHORT).show()
        })
}
```

When the user presses the call button we check if we already got the permission and if not we initiate the previously created launcher supplying it with the permission it needs to ask for. The system remembers the user approval but he can always revoke it and that is why before performing the operation we must always check if we have the permission. (please note that we use the **AppCompat** functions in order to support Android version earlier the 6.0)

```
binding.callBtn.setOnClickListener { it: View!

    if(ActivityCompat.checkSelfPermission(context: this, Manifest.permission.CALL_PHONE)
        != PackageManager.PERMISSION_GRANTED)
        callPermissionLauncher.launch(Manifest.permission.CALL_PHONE)
    else call()
}
```

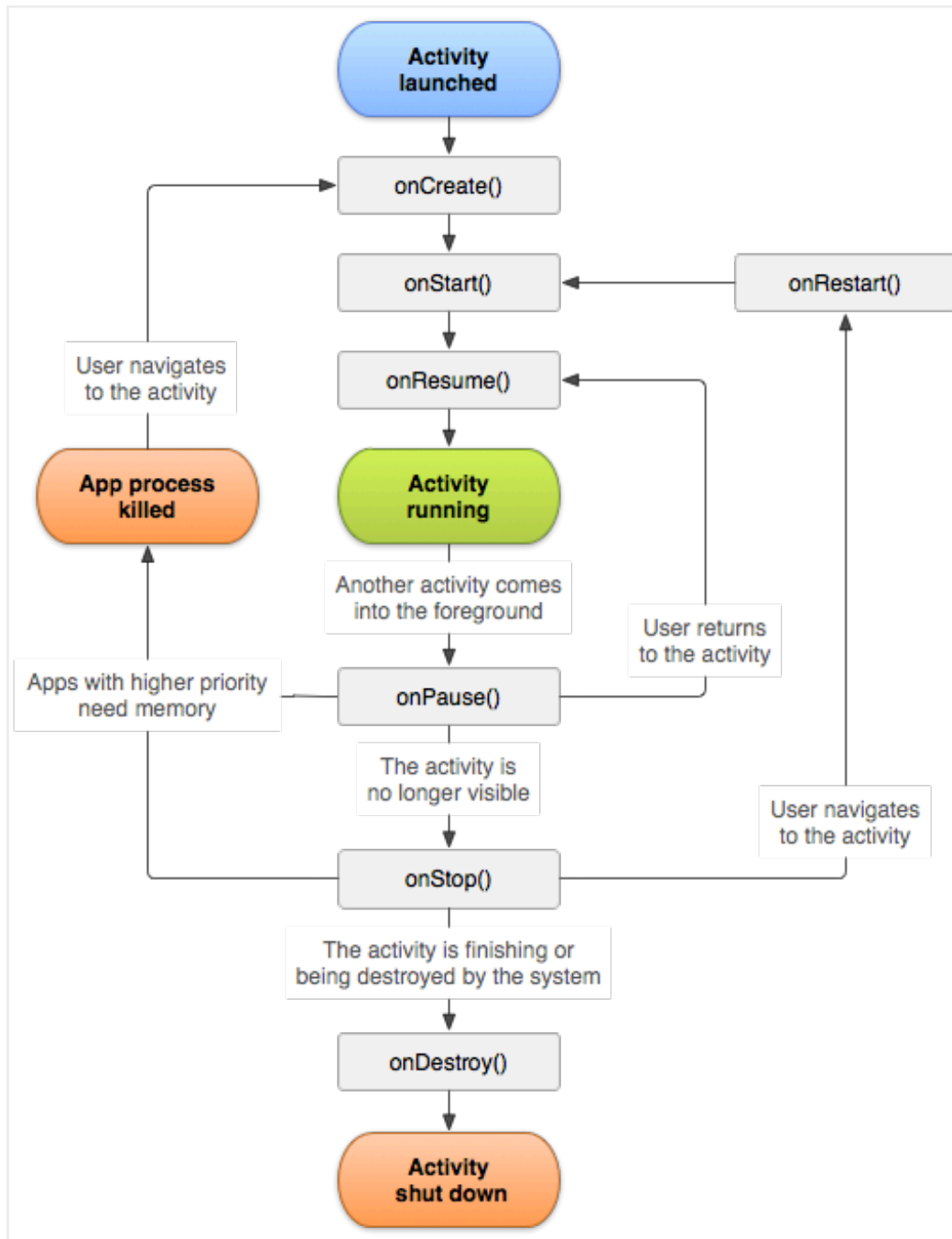
Please note: before asking for the permission Android encourage you to check whether you should show A UI explaining why you need this permission. You can check with the system whether you need to show the rationale with the **shouldShowRequestPermissionRationale()** function if it returns true show a dialog explains why you need it if not just go ahead and ask for it. shouldShowRequestPermissionRationale method returns false only if the user selected *Never ask again* or device policy prohibits the app from having that permission

Activity LifeCycle

An activity has essentially four states:

- If an activity is in the foreground of the screen (at the top of the stack), it is *active* or **running**.
- If an activity has lost focus but is still visible (that is, a new **non-full-sized** window has a focus and it is placed on top of your activity), it is **paused**. A paused activity is completely alive (it maintains all state and member information and remains attached to the window manager), but can't receive interactions from the user.
- If an activity is completely obscured visually by another activity, it is **stopped**. It still retains all state and member information, however, it is no longer visible to the user so its window is hidden and it will often be killed by the system when memory is needed elsewhere.
- If an activity is paused or stopped, **the system can drop the activity from memory** by either asking it to finish, or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state.

The following diagram from the Android Developers shows the important state paths of an Activity. The square rectangles represent callback methods you can implement to perform operations when the Activity moves between states. The colored ovals are major states the Activity can be in.



(Photo from the Android Developers

<https://developer.android.com/guide/components/activities/activity-lifecycle>)

There are three key loops you may be interested in monitoring within your activity:

- The **entire lifetime** of an activity happens between the first call to `onCreate(Bundle)` through to a single final call to `onDestroy()`.
- The **visible lifetime** of an activity happens between a call to `onStart()` until a corresponding call to `onStop()`. During this time the user can see the activity on-screen, though it may not be in the foreground and interacting with the user. Between these two methods you can maintain resources that are needed to show the activity to the user.
- The **foreground lifetime** of an activity happens between a call to

`onResume()` until a corresponding call to `onPause()`. During this time the activity is in front of all other activities and interacting with the user. An activity can frequently go between the resumed and paused states. For example when a dialog indicating a new message arrived, a call received, or any other window that is in the foreground even if it is not fully hides out activity.

In other words: When another window hide even a part of our activity the function **onPause** is called - This function is the best place to save user info to persistent storage.

Note: When overriding each function it is very important to call super first.

Lets examine a situation where we move from activity A to activity B what do think the order of events should be? Think about it.

The key is to remember that while **onPause** is called on the first lost of foreground, `onStop` will only get called when our views are no longer visible, and that will happen only when activity B has the foreground. This is why the order of events will be:

A - `onPause()`
B - `onCreate()`
B - `onStart()`
B - `onResume()`
A - `onStop()`

This is also a good reason to save the data on the `onPause` - if we need it in one of the new activity lifecycle events.

Persistent storage

As we have seen, when the android system calls the **onDestroy()** function all the app memory is deallocated and its resources are freed. So if we need to save some information across the user sessions we can use the lifecycle events to persist data across sessions. Android provides several options for you to save and persist your application data. The solution you choose depends on your specific needs:

- **Shared Preferences** - Store private primitive data in key-value pairs. As its name suggest this is mainly useful in saving simple user preferences like if it is the first run our not, whether he muted the music, whether he want green or blue background color, his already typed e-mail in an edit text and other basic user user information.
- **Internal Storage** - Store private data on the device memory, this storage is designed to be the app "sandbox", its private to your app and will be deleted when the user uninstall your app. It is not limited in size but the user can clear it from the settings. The shared

preferences mentioned above are saved here as well as the ROOM DATABASE we will learn later on, but we can also write to this area directly using Java's streams.

- **External Storage** - Store public data on the shared external storage this information can be shared with other apps and can be saved even after your app is deleted. The External storage called "external" because you can share it with others, **it is not external to the device but to the app**. It is divided to two sections: The first is for use by our app. Like said it can be shared with others. It will be deleted when the user uninstall the app and writing to it doesn't requires permissions. The second is the external shared by all apps, writing and reading from it requires permissions and data saved there will not be deleted when the user uninstall our app.

shared preference

The **SharedPreferences** class provides you the easiest way of saving data to the device memory, the data will be saved while you app is installed on the device. We save all the information with the key-value bundle(hash Table) we have seen before - but with less options - only java primitives, String and a set of Strings.

To get a **SharedPreferences** object for your application, use one of two methods:

- **getSharedPreferences()** - Use this if you need multiple preferences files identified by name, which you specify with the first parameter (use a constant). Use this method to get a preference file to be used across all activities, meaning a file that can be accessed from anywhere in your app.
- **getPreferences()** - Use this if you need only one preferences file for your **Activity**. Because this will be the only preferences file for your Activity, you don't need to supply a name.

Here is an example of using the first option in the **onPause** lifecycle event

```
companion object{  
    const val PREFS = "details"  
}
```



```
override fun onPause() {
    super.onPause()

    sharedPreferences(PREFS, MODE_PRIVATE).edit().apply { this: SharedPreferences.Editor!
        putString("user_name", binding.nameInputLayout.editText?.text.toString())
        putString("phone", binding.phoneNumberInputLayout.editText?.text.toString())
    }.apply()
}
```

Please note that writing to the file system can be either synchronous or asynchronous. If you use the **apply()** function on the editor the writing is done later on but if you use the **commit()** function the system holds everything and writes the data to the filesystem now.

When we want to read data from the shared preferences we use the same file name and keys:

```
getSharedPreferences(PREFS, MODE_PRIVATE).apply { this: SharedPreferences!
    binding.nameInputLayout.editText?.setText(getString("user_name", ""))
    binding.phoneNumberInputLayout.editText?.setText(getString("phone", ""))
}
```

Note the MODE_PRIVATE flag which is our only option - the file is only readable by our app - the other modes WORD_READABLE and WORLD_WRITABLE considered to be dangerous and as of API 17 are deprecated (when google moved to SELinux).

Android Building block - Part 2

Download the full App created in this guide

<https://drive.google.com/file/d/1hiATTqbz-KEhcq20dfqWNX4ZEdqG9p3P/view?usp=sharing>

Download the Animations XML Files

<https://drive.google.com/file/d/1lrXBf5dJL7Lu7O3bNNUyHqMBfvYFJLT5/view?usp=sharing>

Single activity architecture

From the Google **I/O 2018**:

"Today we are introducing the Navigation component as a framework for structuring your in-app UI, with a focus on making a single-Activity app the preferred architecture"

Yes, although we can create as many Activities as we want, this not the recommended architecture according to Google. Activity takes the **whole** screen, when creating it the system has to create a new context and switch to it, also it has to create a new window for the Activity's root view and sometimes this can take a while especially if we don't need all of this work.

A long time ago Google introduced a new OS (Android 3.0) just for tablets. The OS included Fragments as her main key feature. Fragments gave us the ability to split our whole screen into a bunch of individual units the can work together and still each one is independent, it has its own Lifecycle events (that corresponds to the hosting Activity lifecycle). Each Fragment has its own Kotlin and XML files and the most important thing is that creating it is much quicker then creating activity since we don't create a new context or an new window for its root view but instead we just add its root view to a specific view container in the Activity layout and it becomes the Fragment host.

Because it was a later addition the Fragments API was added both to the v4 support library (today replaced by the AndroidX) to be used in lower versions of Android and to the android.app package - based on the idea that when enough time will pass we will use only the android.app and won't be needing the support library anymore. But sometimes realty overcomes and today the android.app Fragments are deprecated and we should only use the ones from the AndroidX support library.

Let's go back to the Google I/O, in 2018 they introduced Navigation as a part of the Jetpack tools for clean and reliable android apps. The Navigation component, like the iOS storyboard allows us to create and design in a nice and easy graphical interface all of our app flow in terms of screens and transitions. You can create and see in one place all of your app screens and the flow

between them and it's all done with, what else, Fragments! In fact the only work the Activity is doing is hosting the Fragments and sometimes interacting with app menus.

Fragments

Like said before a fragment has a few important key features: It has its own layout and Kotlin file. This way he his responsible for is own ui and logic. Besides the fact that it makes our code more structural it makes the fragment an individual unit that can be taken to another project with ease. So go ahead and create a new Empty Activity project with an Activity that will use solely asa a container. **This will be an ongoing project so call it ArchitectureProject.**

In it create a new XML file and add a Floating Action Button in the buttom - end of the parent. Your xml should look like this:

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ItemsFragment">

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/floatingActionButton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="16dp"
        android:layout_marginBottom="16dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:srcCompat="@android:drawable/ic_input_add" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

If you're there already, create our second screen, it will use for adding an item that will be shown later on in a list in that our screen so just add the item input fields in the next screen. Each Item will have a title, a description and an image. So go ahead and create your UI, don't forget the finish button. This is the general layout of the xml file.

Before we need to create our Fragments, let's understand it's lifecycle:

The lifecycle of the Activity in which the fragment resides directly affects the lifecycle of the Fragment. Each lifecycle callback of the activity results in a similar callback for each hosted Fragment. For example, when the activity receives `onPause()`, each fragment in the activity receives `onPause()`.

Fragments have a few extra lifecycle callbacks that handle unique interaction with the activity in order to perform actions such as build and destroy the fragment's UI. These additional callback methods are:

`onAttach()` - Called when the fragment has been associated with the activity (the `Activity` is passed in here by the OS). If the Fragment needs the Context after this function he can retrieve it using the `getActivity` or `requiredActivity`

functions.

onCreateView() -Called to create the view hierarchy associated with the fragment.

onViewCreated() - Called immediately after onCreateView. This gives subclasses a chance to initialize themselves once they know their view hierarchy has been completely created. The views aren't attached to their parents yet.

onActivityCreated() -Called when the activity's **onCreate()** method has returned.

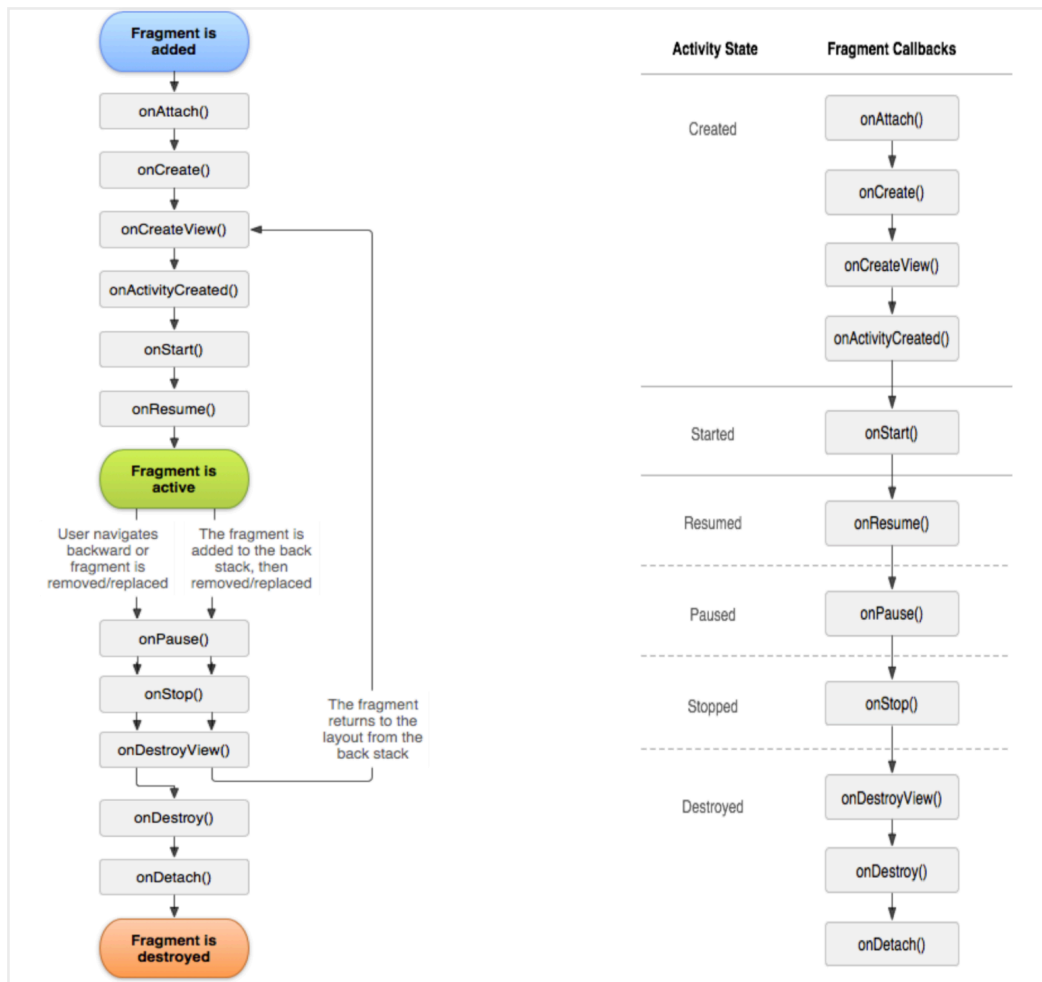
onDestroyView() - Called when the view hierarchy associated with the fragment is being removed.

onDetach() - Called when the fragment is being disassociated from the activity. the getActivity() function here will returned null.

Once the activity reaches the resumed state, you can freely add and remove fragments to the activity. Thus, only while the activity is in the resumed state can the lifecycle of a fragment change independently.

More than that the Fragment's Views has a separate Lifecycle that is managed independently from that of the fragment's Lifecycle.

The fragment views can be destroyed while the fragment itself is alive in the back stack. The back stack designed to imitate the back pressed activity action for fragments - meaning when the user presses the back button the last performed action is popped out. If the action included replacing Fragment A with B then pressing the back button will pop it out and Fragment B will be replaced with A that waited in back stack to be popped out (the instance remained alive while the views weren't). This is important and has affects on the view bidding as we will see soon.



Fragments and the Fragment Manager

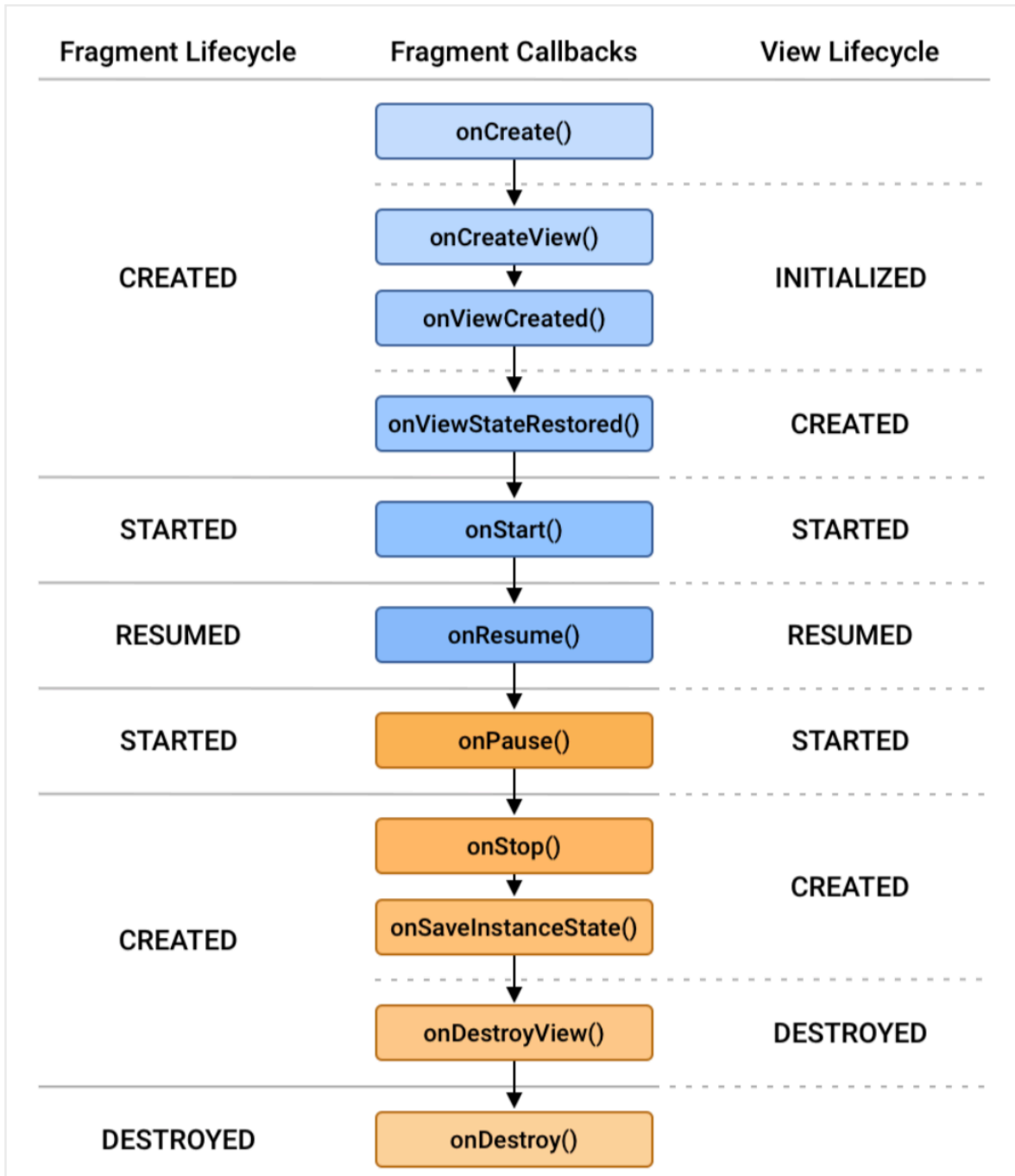
After the **onCreate()** event the fragment is added to the **FragmentManager**. The **FragmentManager** is responsible of attaching fragments to their hosting activity and detaching them. When these events happen the fragments **onAttach()** and **onDetach()** are called. After **onAttach** you can call the **FragmentManager**'s **findFragmentById()** function and get the desired fragment. Besides managing all or our fragments and giving as the ability to add, remove, replace and retrieve them, the **FragmentManager** also manages the back stack we have talked about before.

Like the Activity the Fragment has its own lifecycle and it implements the Jetpack's **LifecycleOwner** interface that allows to retrieve his lifecycle events using the **getLifecycle()** method. This function return a **Lifecycle** object with the following states:

- **INITIALIZED**
- **CREATED**
- **STARTED**
- **RESUMED**
- **DESTROYED**

But don't forget that the fragments keeps a separate lifecycle object for its views in case we need to preform UI related tasks such as start observing data that will only be shown in a list.

Here are the fragment lifecycle events and its view lifecycle events with their corresponding callbacks:



We will see more on those Lifecycle states later on when we dive deeper into Jetpack.

For further reading on the fragment lifecycle please refer to:

<https://developer.android.com/guide/fragments/lifecycle>

Creating our Fragment Kotlin file

First add the viewBinding feature to you app Gradle file:

```
buildFeatures {  
    viewBinding true  
}
```

Now create your Fragments. Inherit from the AndroidX Fragment class and use view binding to inflate our views. Because the views has a separate lifecycle from the fragment itself and it can outlive its views in the back stack, we need to de-allocate our binding object in the **onDestroyView()** method.

For this we have to make a nullable binding field, initiate it in the **onCreateView()** function where we get the layout inflater and the parent, which serves as the fragment container, and after inflating the layout we return the root view. The binding must be assigned null in the **onDestroyView** which causes the GC to de-allocate all the views and release the memory even if the fragment itself is still alive and in this way we can avoid memory leaks. Please note that because it is nullable we create a non-nullable property for easy access which we will use in caution.

```
class ItemsFragment : Fragment() {  
  
    private var _binding : AllItemsFragmentBinding? = null  
  
    private val binding get() = _binding!!  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? {  
        _binding = AllItemsFragmentBinding.inflate(inflater, container, attachToParent: false)  
        return binding.root  
    }  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        super.onViewCreated(view, savedInstanceState)  
    }  
  
    override fun onDestroyView() {  
        super.onDestroyView()  
        _binding = null  
    }  
}
```

Note that when we inflated the Activity layout we didn't supply any parent because the system create a new window just for it, so it's not joining any parent. But here we specify a container since we add its root view to a specific container resides in the hosting activity.

Before going forward to our Navigation component please add the `tools:Context` to each fragment's xml file and reference the Kotlin in order for

the android studio Design to show us our views related to this act fragment.

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ItemsFragment">
```

Adding Navigation for Fragment transactions

Like said before the Fragment Manager is responsible for exchanging and managing Fragments. Each transition can include adding, removing or replacing fragments and is called **Fragment transaction**. In order to imitate the back button press for fragment as it is with Activity (remove the last added screen) a special back stack is created and you can add the Transaction to it. When the user press the back button the last transaction is removed. The fragment can live in the back stack although it's views are destroyed like we said.

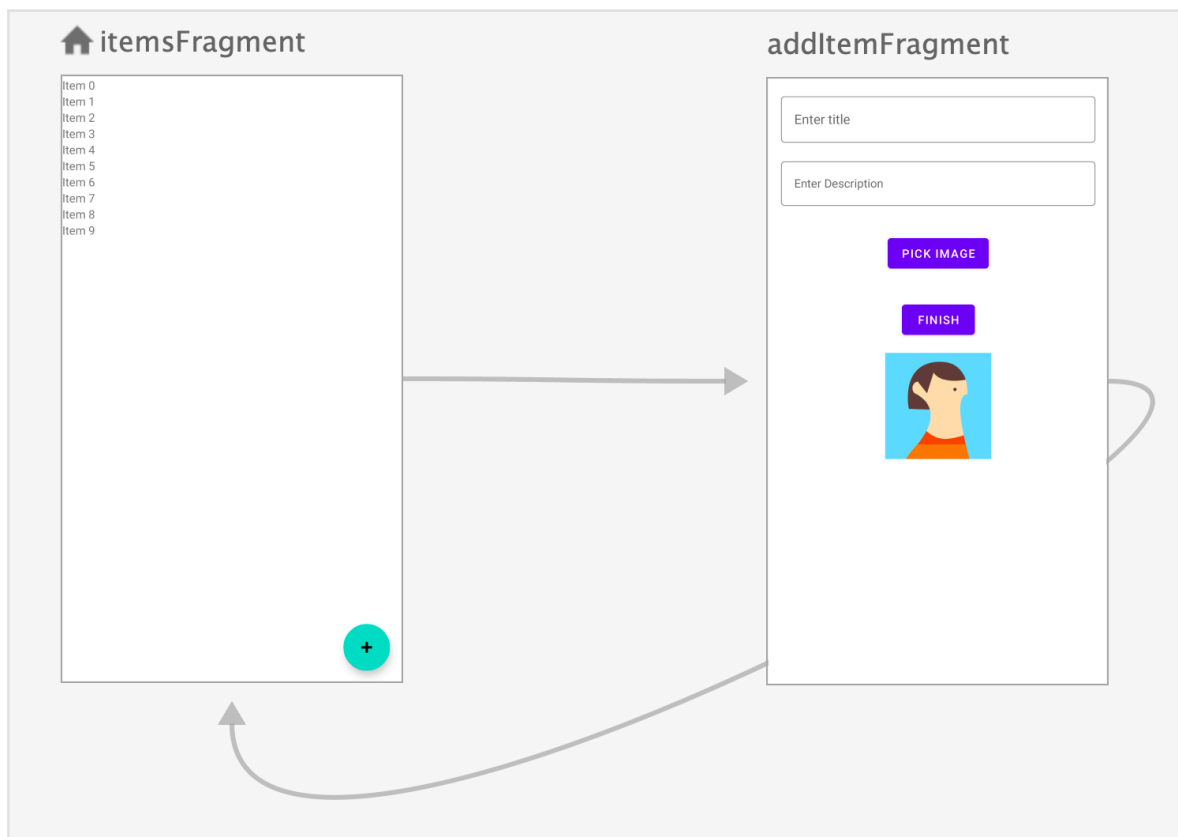
All of this work used to be done manually but as of Google I/O 2018 we can do all of this with the Navigation component.

First we need to add the Navigation graph to our resources and the fragments and their transitions to it. By looking at the graph we will see all of our app screens and the flow between them. We can design our app flow in a very nice and friendly GUI interface and we can even add animations.

So first, implement the following steps:

- Under the res-> new resource. In the dialog choose type Navigation and give it a name. This is your Navigation graph.
- Enter your newly created Navigation xml file and add your fragments. The first fragment you will add will be your home fragment(can be change later on by right clicking on any fragment and setting as home)
- If you can't see the layout in the preview copy the tools from the activity_main.xml file into your navigation xml file and add tools:layout to each fragment and reference his xml file (if you added the tools:context to your root layout of each xml file all should be ok).
- In the design add your actions by dragging from one fragment to another, each arrow added can be executed in our Kotlin code later on - note the addition the the xml file

In the end it will look like that:



By pressing the floating button we will navigate to the adding screen and by pressing the finish button we will go back to our items list.

Next we need need to add the NavHost to your activity.

The navigation host is an empty container where destinations are swapped in and out as a user navigates through your app. When we want to perform our actions we will get a reference to it and execute them. The Navigation host is a simple Layout which inherits from the reliable `FrameLayout` and called **FragmentContainerView**. This Fragment container can create our fragments and execute out fragment transactions. Add it via xml to your root layout - general activity_main.xml file.

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:defaultNavHost="true"
    app:navGraph="@navigation/my_nav" />
```

The **defaultNavHost** property tells the system to pass the back clicks to this NavHost so he can pop his back stack.

Optional you can use the "tag" attribute if you want later to reference him using the Fragment Manager `findFragmentByTag()`.

```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/my_nav" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

So now all we have to do is execute the action and pass extra information with some of them.

To perform these actions we need to get a reference to our Navigation controller.

We can do this with any child view in the its view hierarchy to by:

```
Navigation.findNavController(v).navigate([Action id])
```

Where *v* is any view in a view tree that its root is the Navigation Controller, meaning any view from the displayed Fragments.

If you do not have a live view you can also pass the context and a view id:

```
Navigation.findNavController(this,R.id.text_view).navigate([Action id])
```

But the best and shortest is With the navigation-fragment-ktx library (which already added to your Gradle):

```
From fragment: findNavController().navigate([Action id]);
```

```
From activity: findNavController(R.id.text_view).navigate([Action id])
```

And in our case:

```
binding.finishBtn.setOnClickListener { it: View!
    findNavController().navigate(R.id.action_addItemFragment_to_itemsFragment)
}
```

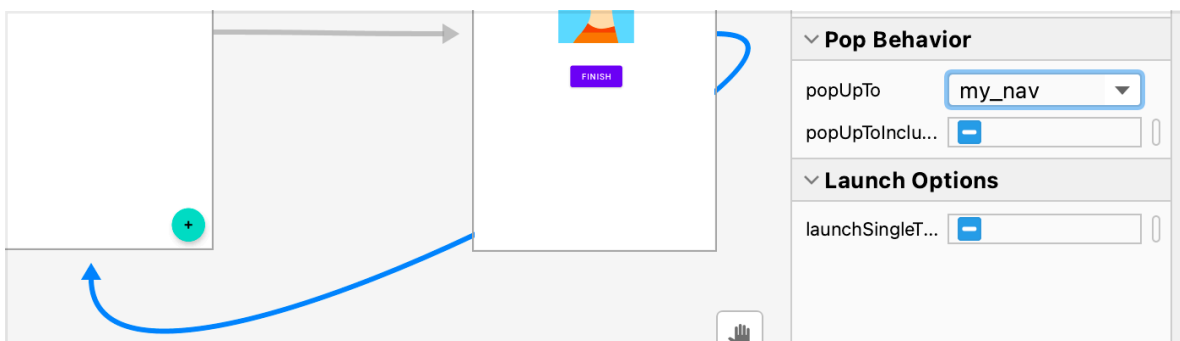
```
binding.floatingActionButton.setOnClickListener { it: View!
    findNavController().navigate(R.id.action_itemsFragment_to_addItemFragment)
}
```

Back stack

Now run the app navigate to the add item screen, press the finish button and go back to the Home Screen. So far so good.

But press the back button. Strange ah? Not so much. By default each action(which is a transaction) is added to the back stack, pressing the back button pop the last action.

You can solve this by pop the back stack with the action. Choose the action In the navigation and in the pop behavior choose the root navigation container. This mean that when executing the action all the fragments in the back stack will get pop up to the very root of the navigation.



Or alternatively you can pop to the home screen but if you do so check the inclusive check box to also pop the former instance of it from thee (otherwise you will have to home screens)



Passing data between Fragments

Each Fragment has an Arguments property which is a bundle and is generally used to pass information to the fragment upon its creation. A common Fragment factory method will receive the data add it to a bundle and set it as the Arguments property of the newly created Fragment it returns. That way this

factory function create a new fragment with the data it needs already inside it. Later on, when we need to get the data it can access its arguments property the get it. The same is done by the our Navigation Controller, when we wants to navigate to a specific fragment and pass some data, we create a bundle and send it with the action to the **navigate()** function. It will automatically set this bundle as the Arguments property of the new Fragment.

Let's pass the item details (not an object and without the photo yet) and show them in a Toast message, for now. Next stage we will create a dynamic list in the all items screen and add the object to it.

Let's create a bundle with the details and call the navigate function with it as a parameter.

```
binding.finishBtn.setOnClickListener { it: View!  
    val bundle = bundleOf( ...pairs: "title" to binding.itemNameEt.text.toString(),  
        "description" to binding.itemDescriptionEt.text.toString())  
    findNavController().navigate(R.id.action_addItemFragment_to_itemsFragment, bundle)  
}
```

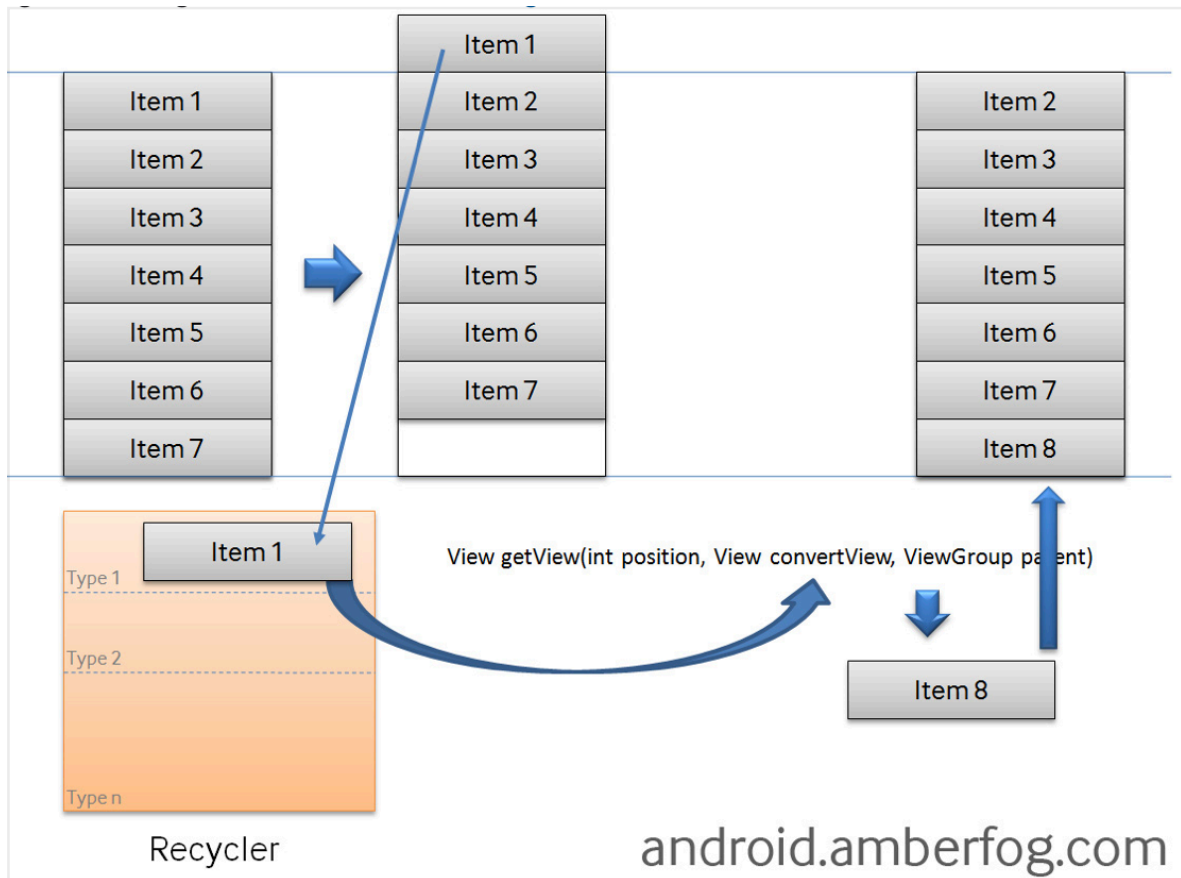
In the newly created fragment that has this bundle in his arguments property we get the data

```
arguments?.getString( key: "title")?.let { it: String  
    Toast.makeText(requireActivity(), it, Toast.LENGTH_LONG).show()  
}
```

RecyclerView & Adapter

A RecyclerView dynamic scrollable list of items. The list is populated dynamically. It is not loading all the cells in advance, but instead it gets them at runtime when the user scroll to it. This is why it uses a Recycle bin like its name suggest. The purpose of the Recycle bin is create a faster scrolling by minimizing the amount of object allocation. In fact the recycler is only creating the amount of initial cells shown to the user and maybe one more. Once the user start scrolling the list the item that is no longer visible is not de-allocated from the heap but instead moves to the recycle bin and when a new cell with the same type (same layout and views) as the old one needs to be created it simply recycle the old one with the new content. This idea is based on the principle on which the content of the cells is different their views isn't so we can simply take an old cell and populate it with the relevant data.

Take a look in the following diagram:



Although what you see here is the old ListView getView() function and the idea is the same.

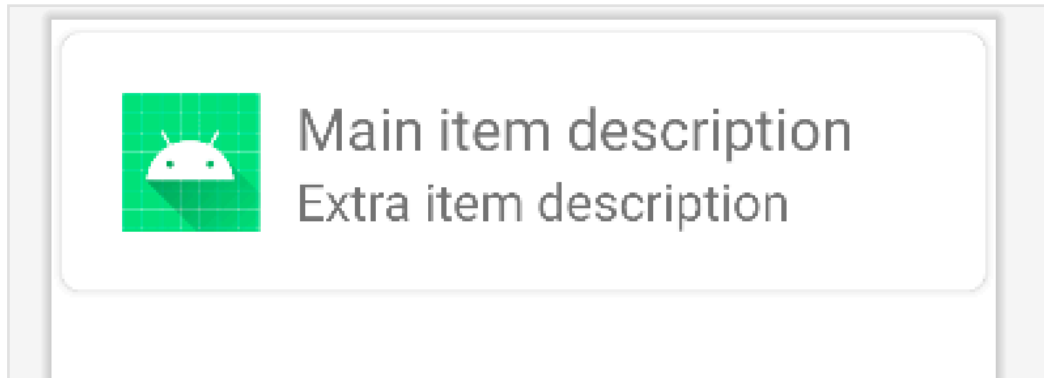
The old item moves to recycle bin and when a new cell should enter if it is from the same type(same layout) as the one in recycle bin the list uses it instead of creating a new one.

So let's use this beautiful mechanism in our project:

First add a RecyclerView to the all_items_layout make it take all the parent space and give it an id

```
<androidx.recyclerview.widget.RecyclerView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/recycler"/>
```

Next design your cell's layout. With RecyclerView we use CardView. So create a new xml with the CardView as the root and design your layout:



To achieve this layout add **contentPadding** and **cornerRadius** while setting the **cardUseCompatPadding** to true in your CardView attributes. This will make nice separation between the cards. Inside the card add an Horizontal linear layout with and image and a vertical linear layout with two TextViews. Don't forget to give each view an id.

Now add the Item data class

```
data class Item(val title:String, val description: String, val photo: String?)
```

Note the the photo property is nullable since not all items will have a photo(at least not in the beginning)

Create an ItemManager object declaration that will serve as a Singleton that holds a list of the items and a functions to add and remove an item to and from the list. Later on we will move the data to the ViewModel and persist it in the local storage with Room database.

So add also this object declaration:

```
object ItemsManger {  
  
    val items : MutableList<Item> = mutableListOf()  
  
    fun add(item: Item) {  
        items.add(item)  
    }  
  
    fun remove(index:Int){  
        items.removeAt(index)  
    }  
}
```

Adapter

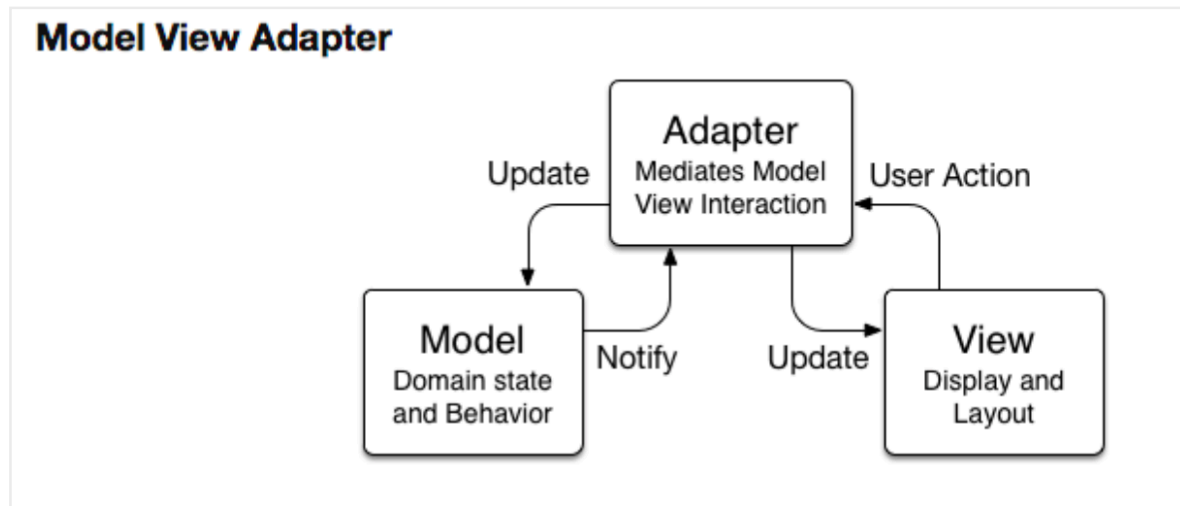
Now that we have both the cell layout the the Kotlin data class we can create an Adapter that connects them together and supply populated views to the recycler.

But First let's understand the concept of the Adapter. According to the MVC design pattern the the controller is a mediator unit between the views and the model in order to separate between the logic and the UI. The MVA (Model View Adapter) is very similar.

Model-View-Adapter is a variation of the Triad where all communication between Model and View must flow through the Adapter, instead of interacting directly as in a Traditional MVC Triad. The Adapter becomes a communication hub, accepting change notifications from Model objects and UI events from the View.

This approach might appear excessively strict, but has some advantages: the communication network is artificially constrained, making it easier to evaluate and debug. All action happens in the Adapter, and the View can be created from off-the-shelf widgets without any Model-specific variation which make him more generic.

MVA is an implementation of the Mediator pattern. Controllers are generally referred as Adapters or Mediators. The Model and the View do not hold references to each other, they do not exchange data nor interact directly.



Create the Adapter - add a new Kotlin class and name it ItemAdpter it should inherit from the **RecyclerView.Adapter** but first let's create our View Holder. A View Holder is like its name suggest a class the holds references to all of our cell's views and given the data class it will bind the data to the views. Our View Holder should inherit from the **RecyclerView.ViewHolder** receive the binding object in its constructor and pass the root to his parent. Then given a data object it will bind the views to their data. So our View Holder should look like this:

```

class ItemAdapter {

    class ItemViewHolder(private val binding: ItemLayoutBinding)
        : RecyclerView.ViewHolder(binding.root) {

        fun bind(item: Item) {
            binding.itemTitle.text = item.title
            binding.itemDescription.text = item.description
        }
    }
}

```

To complete our class definition, define a primary constructor receiving the Items and inherit from the RecyclerView.Adapter (use our ItemViewHolder for the generic view holder). To get rid of the not implementing compilation error press ctrl+I and implement the three abstract functions. Our Adapter should look like this :

```
class ItemAdapter(val items:List<Item>) : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {
        TODO( reason: "Not yet implemented")
    }

    override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
        TODO( reason: "Not yet implemented")
    }

    override fun getItemCount(): Int {
        TODO( reason: "Not yet implemented")
    }
}
```

Let's explain a bit about how the adapter is working:

It responds to the RecyclerView requests. First of all the RecyclerView gets the amount of items by calling the **getItemCount()** function and if the amount is bigger than zero, it asks for them one by one from its Adapter. Now, notice there are two functions for this: the create function and the bind function. As I explained before the first cells displayed on the screen (+one more) needs to be created from scratch, so for them the recycler calls both **onCreateViewHolder** supplying himself as the parent and a type (like we said this is used in case where one recycler cells has more then one layout file), the function returns the newly created ViewHolder, and with it, the recycler calls the **onBindViewHolder** passing the already created empty view holder and the relevant position, and this function use view holder's bind function with the specific Item at the requested position and return cell with the relevant data so he can Add it to the list and show the user. But as we said before when the user start scrolling the scrolled out cell moves to the recycle bin and then the recycler doesn't have to call the "expensive" **onCreateViewHolder** but only the "cheap" and fast **onBindViewHolder** function. So it's the recycler choice when to create the cell or just bind the data according to what it has in its recycle bin.

So out full Adapter code should look like this:

```
class ItemAdapter(private val items:List<Item>) : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        ItemViewHolder(ItemLayoutBinding.inflate(LayoutInflater.from(parent.context),parent, attachToParent: false))

    override fun onBindViewHolder(holder: ItemViewHolder, position: Int) = holder.bind(items[position])

    override fun getItemCount() = items.size

    class ItemViewHolder(private val binding: ItemLayoutBinding)
        : RecyclerView.ViewHolder(binding.root) {

        fun bind(item: Item) {
            binding.itemTitle.text = item.title
            binding.itemDescription.text = item.description
        }
    }
}
```

Later we will get back to it and update the image as well as handling view events, but for now it's enough.

Connecting the Recycler to the Adapter and setting the Layout Manager

What we need to do now is to connect between the Recycler and the Adapter but before that we need to set a Layout Manager to the Recycler View.

A Layout manger decides how the cell will be organized. We have three options:

1. **LinearLayoutManager** - organizes the cells one after the other like a scrolling list, it can be either horizontal or vertical.
2. **GridLayoutManager** - organizes the cells in a grid or a table where we must supply number of columns.
3. **StaggeredGridLayoutManager** - is the same as before but each square in the grid can have a different height (like the notes app)

We will use the Linear Manager. So in the onCreateView or onViewCreated in All Items Fragment we set adapter and the layout manager of the Recycler view we have added before to the xml file. In the **onViewCreated** after setting the layout manager pass the List of items from the ItemManger object to the Adapter's constructor and set him is our recycler view's adapter. So our almost finished code should look like this:

```
override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)

    binding.recycler.layoutManager = LinearLayoutManager(requireContext())
    binding.recycler.adapter = ItemAdapter(ItemsManger.items)
}
```

Before we run the app and see how the magic happens we first need to add items to the list. So add the new item to the ItemManger object and remove the bundle from the navigate action. Your code should look like this:

```
binding.finishBtn.setOnClickListener { it: View!

    val item = Item(binding.itemNameEt.text.toString(),
        binding.itemDescriptionEt.text.toString(), photo: null)
    ItemsManger.add(item)

    findNavController().navigate(R.id.action_addItemFragment_to_itemsFragment)
}
```

Run your app and test your recycler. It's working very nicely but be aware of the fact that the list not saved to the file system.

Receiving events from Recycler

First of all, unlike other Views the Recycler doesn't have any interface throughout which it can send us user events.

Having said that, If all you want is dragging and swiping you have a pre-made Helper you can attach recycler like this:

```

binding.recycler.layoutManager = LinearLayoutManager(requireContext())
binding.recycler.adapter = ItemAdapter(ItemsManger.items)

ItemTouchHelper(object : ItemTouchHelper.Callback() {

    override fun getMovementFlags(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder
    ) = makeFlag(ACTION_STATE_SWIPE, directions: LEFT or RIGHT)
        //or makeFlag(ACTION_STATE_DRAG, UP or DOWN or LEFT or RIGHT)

    override fun onMove(
        recyclerView: RecyclerView,
        viewHolder: RecyclerView.ViewHolder,
        target: RecyclerView.ViewHolder
    ): Boolean {
        TODO(reason: "Not yet implemented")
    }

    override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int) {
        ItemsManger.remove(viewHolder.adapterPosition)
        binding.recycler.adapter!!.notifyItemRemoved(viewHolder.adapterPosition)
    }
}).attachToRecyclerView(binding.recycler)

```

The **getMovementFlags()** return the available gestures (swiping and / or dragging) and to and from which direction.

The **onMove()** function is called upon dragging event

And the **onSwiped()** upon swiping. Since it's all we allow here, we implement only it and remove the item from the data. But the adapter which also has a reference to this list should know that its data source has changed and it needs to notify the recycler to get updated and read the data again. There are few functions throughout which the adapter can cause the recycler to get updated one is **notifyDataSetChanged()** which causes the recycler to read all of the adapter data all over again but if have a more specific change we can use the **notifyItemRemoved/Inserted/Updated** and pass them the exact index of the update. Besides the fact that it's more efficient it's also done with Animation.

Receiving custom events from our RecyclerView

This is a bit more tricky. Because when we want get a custom event we need to attach a listener to the Views, but the only one who has access to these views is the adapter and if we write the event handing code in the adapter we decide on one implementation for all and loose our ability to be generic and let the class using the Adapter decide of its own event handling. Like here where the fragment is implementing the **onSwiped()**.

We want to create exactly this. We want to separate the event from the event handling, and let another class decide for itself on how to respond to the event

we are reporting about.

So first decide on the events you want to report about and the info you want to send with it and create your interface inside the Adapter:

```
class ItemAdapter(private val items:List<Item>){  
  
    interface ItemListener {  
        fun onItemClick(index: Int)  
        fun onItemLongClick(index: Int)  
    }  
}
```

Make the Adapter's constructor to receive an instance of that Listener (as well as the list) and make the View Holder class (the one receiving this actual events) to be an **inner** class so it can access this callback and invoke its functions (the functions that will be later implemented in the fragment for example).

```
class ItemAdapter(private val items:List<Item>, private val callback : ItemListener)  
    : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {
```

Our View holder should register for these view events and call the event callback function with the relevant info:

```

inner class ItemViewHolder(private val binding: ItemLayoutBinding)
: RecyclerView.ViewHolder(binding.root),
View.OnClickListener, View.OnLongClickListener {

    init {
        binding.root.setOnClickListener(this)
        binding.root.setOnLongClickListener(this)
    }

    override fun onClick(p0: View?) {
        callback.onItemClicked(adapterPosition)
    }

    override fun onLongClick(p0: View?): Boolean {
        callback.onItemLongClick(adapterPosition)
        return true
    }

    fun bind(item: Item) {
        binding.itemTitle.text = item.title
        binding.itemDescription.text = item.description
    }
}

```

In in the Fragment just implement this functions when you create the Adapter and show a Toast message:

```

binding.recyclerView.layoutManager = LinearLayoutManager(requireContext())
binding.recyclerView.adapter = ItemAdapter(itemsManger.items, object : ItemAdapter.ItemListener {
    override fun onItemClicked(index: Int) {
        Toast.makeText(requireContext(), text: "${itemsManger.items[index]}", Toast.LENGTH_SHORT).show()
    }

    override fun onItemLongClick(index: Int) {
        TODO("reason: \"Not yet implemented\")
    }
})

```

Last step - add the photo from the gallery

What we want to do is for the user to pick an image from the gallery (later on we will add also the camera option). This is part of the Start Activity For Result API which creates a new Activity and returns its result.

Let's go back the Runtime Permissions, the same mechanism is implemented here. In fact the runtime permission was just a single use case of the entire Launchers API mechanism. This mechanism is discussed in details in its own module. In short the idea behind it is to register a launcher in the activity or

fragment creation and when we need to, launch it - the idea is to make the call and the result not dependent in each other.

Here we use a different contract then the request permission contract, the contract will be the **OpenDocument()** contract in which the launcher receives an array of strings representing the mime types of the files we want to show to the user to choose one from. The result is the Uri of the specific file chosen by the user. Because we launch another component which reads the file storage and display it to the user for him a choose from, we don't need to ask for the reading permission ourself. Instead the activity that actually read the storage should ask for the permission.

So add the launcher creation to the AddItemFragment and in the callback that receives the URI of the photo chosen, set this photo in the image view and add it to the Item instance.

Be aware of the fact that for security reasons this Uri is temporary it is valid until our activity session will end(until **onDestroy()**). Because we need to save it in the file system later on, we need to ask for the OS to make the Uri persistent. This is done through the Content Resolver component that will be discussed later on as well as the actual saving of the item in the local DB.

Our Launcher definition will look like this:

```
private var imageUri : Uri? = null

val pickItemLauncher: ActivityResultLauncher<Array<String>> =
    registerForActivityResult(ActivityResultContracts.OpenDocument()) { it: Uri!
        binding.imageView.setImageURI(it)
        requireActivity().contentResolver.takePersistableUriPermission(it, Intent.FLAG_GRANT_READ_URI_PERMISSION)
        imageUri = it
    }
```

Replace the phot null value with the imageUri in the Item constructor call

```
val item = Item(binding.itemNameEt.text.toString(),
    binding.itemDescriptionEt.text.toString(), imageUri.toString())
ItemsManger.add(item)
```

In the Pick Photo button click launch you launcher and give the "image/*" mime type which means images of all types

```
binding.picButton.setOnClickListener { it: View!
    pickItemLauncher.launch(arrayOf("image/*"))
}
```

Now we see the photo in the ImageView but not yet in the recycler. To achieve this we need to go back to our bind function of the View Holder and use the external Open Source Glide library to read the image from the Uri stored in The

item class into the image view of the cell. The reason we use Glide is besides of its incredible images caching and auto resizing that greatly improves performance it is also doing all of its IO work on a background thread automatically and update the UI on the main thread - we don't need to worry about it - it is also done very efficiently.

So add the Glide dependency to the App grade file and sync your project (you can find the latest in the [Glide GitHub](#))

```
implementation 'com.github.bumptech.glide:glide:4.12.0'  
annotationProcessor 'com.github.bumptech.glide:compiler:4.12.0'
```

And in the bind function use it to load the image and make it nice and round into the image view:

```
fun bind(item: Item) {  
    binding.itemTitle.text = item.title  
    binding.itemDescription.text = item.description  
    Glide.with(binding.root).load(item.photo).circleCrop().into(binding.itemImage)  
}
```

That's it our project if finished for now. Later on we will move our data to the View Model, notify about changes in it with the Live Data and make it persistent with ROOM database.

APPENDIX - Parcelable and Serializable

If we need to pass the object from the Adding Fragment to the All Items Fragment we must put it in the bundle and sends it with the navigation. Please note that we can't put an object reference in the bundle since it is generally used to pass data between components(activities for example) and sending object references between Android components is not possible because by changing the process, the object references won't be in the new process, so we must make our objects Parcelable or Serializable. It means turning them into streams of bytes and put it in the the Bundle. Serializable is simpler since it doesn't require implementing any methods but with more overhead since all the work is done at runtime and reflection in general cost more in terms of efficiency so we would rather use Parcelable because its is built for that exact purpose and is highly optimized for IPC (Inter Process Communication). But there are allot of functions to add so we rather use the kotlin-parcelize plugin in the app Gradle file:


```
ent.kt x build.gradle (:app) x ac
n use the Project Structure dialog to v
plugins {
    id 'com.android.applicat
    id 'kotlin-android'
    id 'kotlin-parcelize'
}
```

This plugin along with the @Parcelize Annotation in the class definition will cause the compile to generate all the Parcel functions for us

```
@Parcelize
data class Item(val title:String, val description: String, val photo: String?) : Parcelable
```

Another option without any plugin or annotations is to implement the Serializable interface which all of his functions are added in runtime

```
data class Item(val title:String, val description: String, val photo: String?) : Serializable
```

And that's it, now we can send it in the Bundle and add it to the list. But the adapter which also reference this list should know that it's data source has changed and it needs to notify the recycler to get updated and read from it the most relevant data. There are few functions through which the adapter can cause the recycler to get adapted one is **notifyDataSetChanged()** which causes the recycler to read all of the adapter data all over again but if we have a more specific change we can use the **notifyItemRemoved/Inserted/Updated** and pass them the exact index to refresh the view.

So our code for sending the Item in the Add Item Fragment should look like this:

```
binding.finishBtn.setOnClickListener { it: View!
    val bundle = bundleOf( ...pairs: "item" to Item(binding.itemNameEt.text.toString(),
                                                binding.itemDescriptionEt.text.toString(),
                                                photo: null))
    findNavController().navigate(R.id.action_addItemFragment_to_itemsFragment,bundle)
}
```

ViewModel & LiveData

Download the full App created in this guide

<https://drive.google.com/file/d/19hr4KqbyGgZvhjx1M4rBZIZcxzJ5wbm/view?usp=sharing>

Download the Navigation View Model app from this guide

https://drive.google.com/file/d/144foCUITej6Amnw4M9Zpa1o2dXIdePr_/view?usp=sharing

Steps:

1. Create a new project use the Empty Activity template
2. Make sure in your project structure dependencies that the view model and live data are included as you can see in the photo attached. If not Search for ViewModel and copy the latest dependencies into your app Gradle file and sync. Copy also the LiveData dependencies we will need it later on. Here is s link to https://developer.android.com/jetpack/androidx/releases/lifecycle#declaring_dependencies

```
androidx.lifecycle:lifecycle-common:2.3.1
androidx.lifecycle:lifecycle-livedata:2.0.0
androidx.lifecycle:lifecycle-livedata-core:2.3.1
androidx.lifecycle:lifecycle-runtime:2.3.1
androidx.lifecycle:lifecycle-viewmodel:2.3.1
androidx.lifecycle:lifecycle-viewmodel-savedstate:2.3.1
```

3. In the Activity layout file select the existing TextView widget and use the Attributes tool window to change the id property to result_text. Drag a Number (Decimal) view from the palette and position it above the existing TextView. With the view selected in the layout refer to the Attributes tool window and change the id to dollar_text. Drag a Button widget onto the layout so that it is positioned below the TextView, double-click on it and to edit the text and change it to read "Convert". With the button still selected, change the id property to convert_btn. Click on the Infer constraints button to add any missing layout constraints. If you don't want to bother yourself with ui now, you can copy the activity_main.xml file located in our starter files.
4. Now add you Kotlin code to preform the conversion and show it in the result Text View.
5. When all is done execute your program and check it.
6. Well done - now please rotate your screen - What Happened??

Our complete code so far should look like this:

```
class MainActivity : AppCompatActivity() {  
  
    companion object {  
        const val EURO_DOLLAR_RATE = 1.17  
    }  
  
    private lateinit var binding : ActivityMainBinding  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        binding = ActivityMainBinding.inflate(layoutInflater)  
        setContentView(binding.root)  
  
        binding.button.setOnClickListener { it: View!  
  
            binding.resultText.text = "${binding.dollarText.text.toString().toDouble().times(EURO_DOLLAR_RATE)}"  
  
        }  
    }  
}
```

Configuration changes

What Happened is that screen orientation changes is a **configuration change** (like language, resolution, text size and much more) in order to load the more specific resources for that configuration the system automatically sends a kill signal to all of the activities and fragments displayed on the screen and creates a new instances - when the new instance is created all the specific resources are loaded by default. That is why all the information is gone. In our case the data is simple but think of cases where we work with REST API or with other remote databases - this requires refetching our data again and again and can affect the user experience.

Old solution

Besides preventing the screen orientation changes in the manifest (by forcing only one orientation for each activity) or telling the system we want to handle this specific change ourselves (also in the manifest - `android:configChanges="orientation"` for each activity and overriding the `onConfigurationChange` methods inside the activities) - This are all ways to bypass the default system behavior. If we we leave the default system behavior that kills the activity, we should have overridden the **onSaveInstanceState** and **onRestoreInstanceState** in the activity and pass the information manually with the outgoing and incoming Bundle.

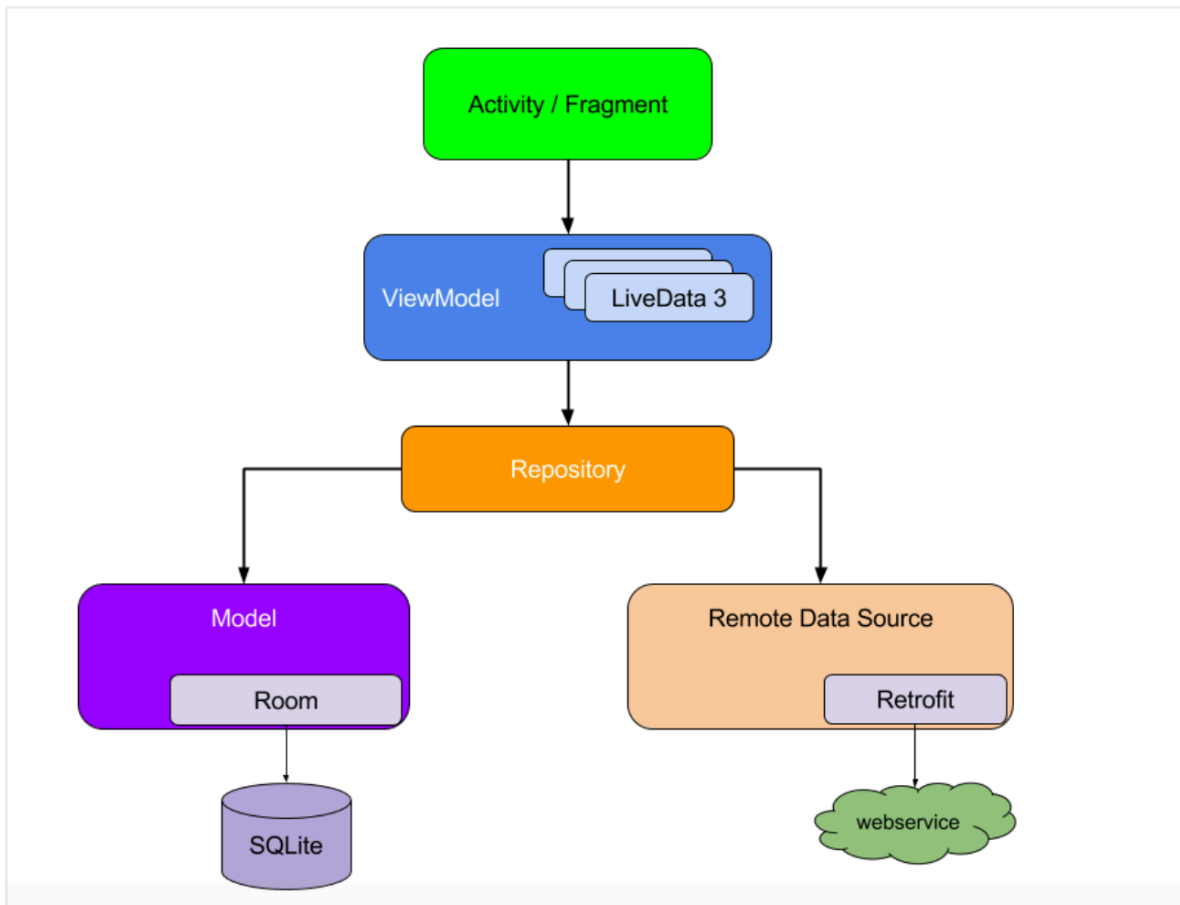
```

override fun onSaveInstanceState(outState: Bundle, outPersistentState: PersistableBundle) {
    super.onSaveInstanceState(outState, outPersistentState)
}

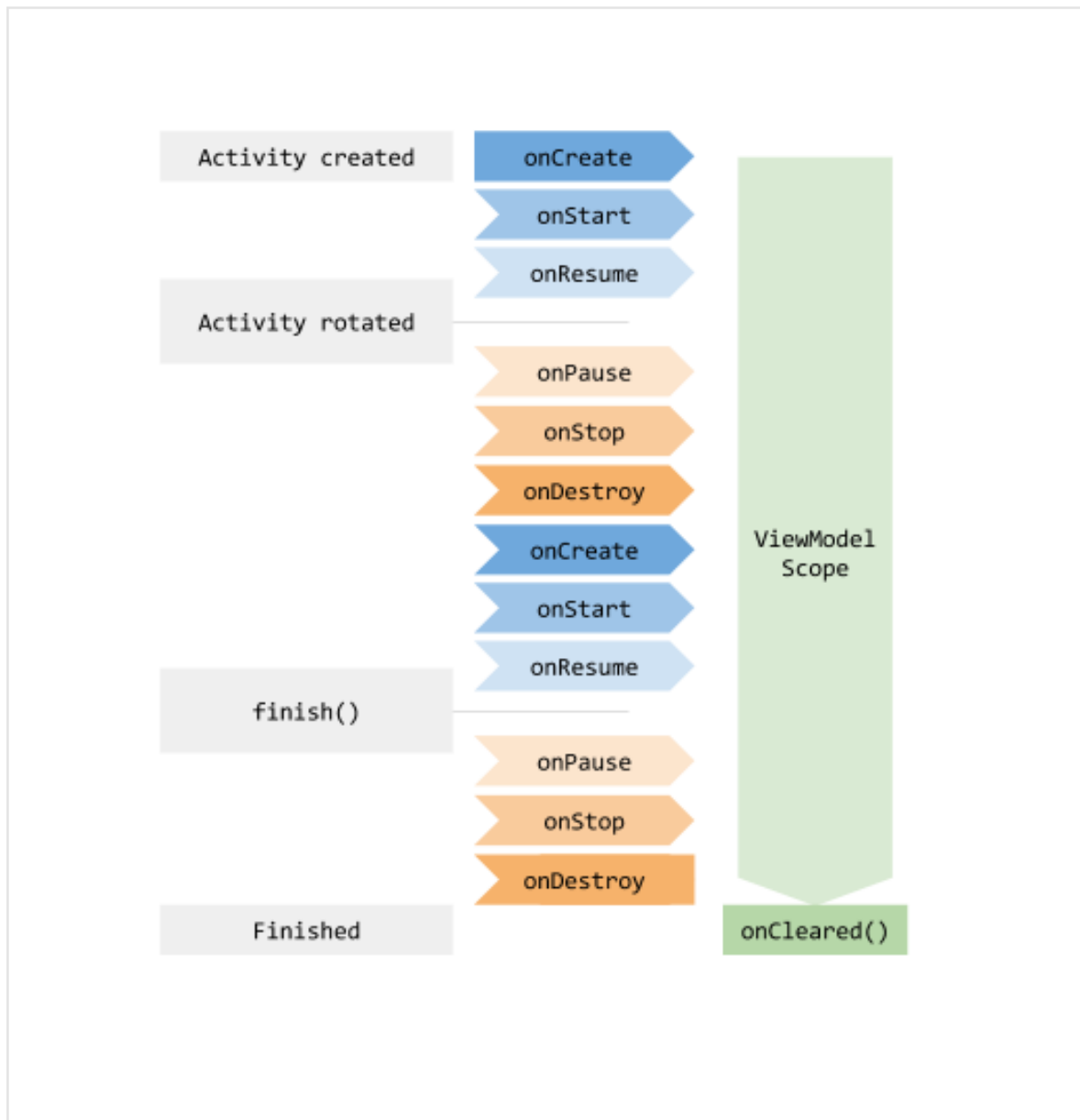
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
}

```

New Solution MVVM - Model - View - ViewModel



Please look at the attached photo from the android developers:



As you can see a ViewModel is a special class that is aware of our lifecycle and can outlive our views (Activities and Fragments). Because of this, it can stay alive during configuration changes. This is the place where we save all of our data. Meaning the activity and fragments will serve only for UI purposes and will not hold any data. Instead, they will have a reference to their specific ViewModel and it will save the data for them and outlives their configuration changes. Only when the activity `onDestroy()` is called without `onCreate()` immediately after it then the `viewModel.onCleared()` function is called and the instance holding our data is deallocated.

Our ViewModel is aware of our lifecycle when we pass `LifecycleOwner` as us and with the help of the `LiveData` it will update us automatically on any change in it or when our new lifecycle event requires it.

Let's implement it:

1. Create a new class and inherit from the **ViewModel** class. Please note

you can also inherit from the **AndroidViewModel** in cases where the Context is needed for example when working with databases.

2. Now we remove all the data and the data related functions to that ViewModel: create a var property called result of type Double and initialize it to 0.0, add a custom setter that receive the value multiply it by the conversion rate and save the result in the field

```
class MainViewModel : ViewModel() {

    companion object {
        const val EURO_DOLLAR_RATE = 1.17
    }

    var result : Double = 0.0
        set(value) {
            field = value * EURO_DOLLAR_RATE
        }
}
```

3. Fragments and Activities needs to obtain a reference to the ViewModel in order to be able to access the model and observe data changes (later on). A Fragment or Activity maintains references to the ViewModels on which it relies for data using an instance of ViewModelProvider class. A ViewModelProvider instance is created via a call to the ViewModelProviders(owner) method from within the Fragment or Activity and pass the current Fragment or Activity as the lifecycle owners. It returns a ViewModelProvider instance. Once the ViewModelProvider instance has been created, the get() method can be called on that instance passing through the class of specific ViewModel that is required - the reflection class file but Use 'java' property to get Java class corresponding to this Kotlin class. The provider will then either create a new instance of that ViewModel class, or return an existing instance.
4. In the button click set the result field in your viewModel and read the updated value to the TextView. Please note that you also need to read it on the onCreate() in case of configuration change (don't worry, we will remove all of this when we use LiveData).
5. Run the app and rotate the screen - The amount saved in the view model and the activity is reading it in any new instance created!

Our complete code should look like this:

```

class MainActivity : AppCompatActivity() {
    |
    private lateinit var binding : ActivityMainBinding

    private lateinit var viewModel : MainViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        viewModel = ViewModelProvider( owner: this)[MainViewModel::class.java]

        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.button.setOnClickListener { it: View!

            viewModel.result = binding.dollarText.text.toString().toDouble()
            binding.resultText.text = viewModel.result.toString()
        }

        binding.resultText.text = viewModel.result.toString()
    }
}

```

Please Note you can also use the KTX extensions to initialize the ViewModel lazy.

Just add the:

```
implementation("androidx.activity:activity-ktx:1.4.0")
```

To your app Gradle file and write the following code:

```

private val viewModel : MainViewModel by viewModels()

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    //viewModel = ViewModelProvider(this)[MainViewModel::class.java]

```

If you want your view model in your Fragment:

```
implementation("androidx.fragment:fragment-ktx:1.4.0")
```

In conclusion this is how to get your View Model with the KTX in activity or fragment:

```
// Get a reference to the ViewModel scoped to this Fragment
val viewModel by viewModels<MyViewModel>()

// Get a reference to the ViewModel scoped to its Activity
val viewModel by activityViewModels<MyViewModel>()
```

Or like before you can mention the variable type and let the generic be inferred from that.

```
val sharedViewModel : MainViewModel by activityViewModels()

val viewModel : MainViewModel by viewModels()
```

LiveData & MutableLiveData

That's all very nice but please note the repetitive code to read the data from the ViewModel.

If we use LiveData or MutableLiveData (if the contained data can change) we can observe the return result and get notified automatically in two cases:

1. The inner value that the LiveData wraps changes.
2. A lifecycle event that requires refreshing the value occurred.

LiveData is a holder class which holds and updates the activity/fragment keeping in mind about their state. It uses special function called **observe** which will update activity/fragment instantly if anything changes in the LiveData. As LiveData class get the latest updated data but couldn't find activity or fragment, then they just hold the data and next time when activity or fragment is resumed the observer will fetch updated data itself and provide it to activity/fragment. For example, an activity that was in the background receives the latest data right after it returns to the foreground.

If an activity or fragment is recreated due to a configuration change, like device rotation, it immediately receives the latest available data. Our observers are bounded to activity or fragment so they will be destroyed when the activity/fragment is destroyed. No need to handle it manually.

LiveData has some characteristics according to Google I/O 2017:

- LiveData is an observable data holder so it can be observed.
- Its lifecycle aware that prevents memory leakage in such a situation like configuration changes.
- LiveData automatically manages subscriptions. If you are observing a LiveData you don't need to unsubscribe. The right things will happen in the right times.
- Doesn't matter how many observers you have or what state they are, all of it are merged into one lifecycle.
- It doesn't have any activity or fragment inside it but it works with both

of them.

- Also LiveData makes testing easy because it's kind of Android free(it can be tested with our device).
- The LiveData instance is doing all the fetching and updating work on the Dispatchers.IO and not on the main thread.

Adding LiveData

1. In the ViewModel class replace the type of the system result from Double to MutableLiveData<Double> - it is mutable since the inside value - the double can change. Remove the get and set and create a new function called setValue that receives the new Double value and update the value field of the LiveData with the converted amount

```
const val EURO_DOLLAR_RATE = 1.17

class MainViewModel : ViewModel() {

    val result = MutableLiveData<Double>()

    fun setValue(value: Double) {
        result.value = value * EURO_DOLLAR_RATE
    }
}
```

2. In the MainActivity remove all of the result Text View updates. Remove also the viewModel update from before. Now in the onCreate() set an Observer to the ViewModel's LiveData field, passing it the activity as the lifecycle owner (for all the reasons mentioned above) and in the callback add the one and only result Text View update.

```
viewModel.result.observe( owner: this) { it: Double!
    binding.resultText.text = it.toString()
}
```

3. Now in the onClick just call the the ViewModel setValue function passing it the user dollar value and Thats it - The LiveData will do the rest!

```
binding.button.setOnClickListener { it: View!  
  
    viewModel.setValue(binding.dollarText.text.toString().toDouble())  
  
    //binding.resultText.text = "${binding.dollarText.text.toString().toD  
    // viewModel.result = binding.dollarText.text.toString().toDouble()  
    // binding.resultText.text = viewModel.result.toString()  
}
```

ViewModel to Communicate between fragments and Their hosting activity

ViewModel is an ideal choice when you need to share data between multiple fragments or between fragments and their hosting Activity.

Lets look at this ItemViewModel which will be shared by both the Activity and its hosted Fragment:

```
class ItemViewModel : ViewModel() {  
    private val mutableSelectedItem = MutableLiveData<Item>()  
    val selectedItem: LiveData<Item> get() = mutableSelectedItem  
  
    fun selectItem(item: Item) {  
        mutableSelectedItem.value = item  
    }  
}
```

Please note that while the actual stored value is MutableLiveData the get only return LiveData this ensures consistency of our data.

Both your fragment and its host activity can retrieve a shared instance of a ViewModel with activity scope by passing the activity into the [ViewModelProvider](#) constructor. The ViewModelProvider handles instantiating the ViewModel or retrieving it if it already exists. Both components can observe and modify this data (in this example we use the KTX extensions library to get a delegate that initial their view model lazy):

```
class MainActivity : AppCompatActivity() {
    // Using the viewModels() Kotlin property delegate from the activity-ktx
    // artifact to retrieve the ViewModel in the activity scope
    private val viewModel: ItemViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewModel.selectedItem.observe(this, Observer { item ->
            // Perform an action with the latest item data
        })
    }
}

class ListFragment : Fragment() {
    // Using the activityViewModels() Kotlin property delegate from the
    // fragment-ktx artifact to retrieve the ViewModel in the activity scope
    private val viewModel: ItemViewModel by activityViewModels()

    // Called when the item is clicked
    fun onItemClick(item: Item) {
        // Set a new item
        viewModel.selectItem(item)
    }
}
```

Share data between fragments

Two or more fragments in the same activity often need to communicate with each other. For example, imagine one fragment that displays a list and another that allows the user to apply various filters to the list.

These fragments can share a ViewModel using their activity scope to handle this communication. By sharing the ViewModel in this way, the fragments do not need to know about each other, and the activity does not need to do anything to facilitate the communication.

```
class ListFragment : Fragment() {
    // Using the activityViewModels() Kotlin property delegate from the
    // fragment-ktx artifact to retrieve the ViewModel in the activity scope
    private val viewModel: ListViewModel by activityViewModels()
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        viewModel.filteredList.observe(viewLifecycleOwner, Observer { list ->
            // Update the list UI
        })
    }
}

class FilterFragment : Fragment() {
    private val viewModel: ListViewModel by activityViewModels()
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        viewModel.filters.observe(viewLifecycleOwner, Observer { set ->
            // Update the selected filters UI
        })
    }
}
```

Notice that both fragments use their host activity as the scope for the ViewModelProvider. Because the fragments use the same scope, they receive the same instance of the ViewModel, which enables them to communicate back and forth.

Saved State module for ViewModel

As mentioned before View Model can survive configuration changes and store our data. Before using it we used to pass our data through savedInstanceState Bundle. We can still use the onSaveInstanceState() as a backup in case our recreation comes from system-initiated process death. In that case our View Models will be killed also.

But onSaveInstanceState() function is from the Activity where the ViewModel actually the one that stores or remembers the UI state so that can cause allot of boilerplate code. To solve this the view model has its own Bundle that can store data between sessions.

All you have to do is to get the SaveStateHandle in you ViewModel's constructor (We will see later on that it also has a default binding)

```
class SavedStateViewModel(private val state: SavedStateHandle) : ViewModel() {
```

Don't worry you don't have to do any additional configuration because the default ViewModel factory provides the appropriate SavedStateHandle to your ViewModel. So just go ahead and retrieve your View model like you did before

```
class MainFragment : Fragment() {
    val vm: SavedStateViewModel by viewModels()

    ...
}
```

The SavedStateHandle class is a key-value map that allows you to write and retrieve data to and from the saved state through the `set()` and `get()` methods. Additionally, you can retrieve values from SavedStateHandle that are wrapped in a `LiveData` observable using `getLiveData()`. When the key's value is updated, the LiveData receives the new value. Most often, the value is set due to user interactions, such as entering a query to filter a list of data. This updated value can then be used to `transform LiveData`.

```
class SavedStateViewModel(private val savedStateHandle: SavedStateHandle) : ViewModel() {
    val filteredData: LiveData<List<String>> =
        savedStateHandle.getLiveData<String>("query").switchMap { query ->
            repository.getFilteredData(query)
        }

    fun setQuery(query: String) {
        savedStateHandle["query"] = query
    }
}
```

By using SavedStateHandle, the query value is retained across **process death**, ensuring that the user sees the same set of filtered data before and after recreation without the activity or fragment needing to manually save, restore, and forward that value back to the ViewModel.

Here is a simple example on how to save the current user in SaveStateHandle

```
class MyViewModel(state : SavedStateHandle) : ViewModel() {

    // Keep the key as a constant
    companion object {
        private val USER_KEY = "userId"
    }

    private val savedStateHandle = state

    fun saveCurrentUser(userId: String) {
        // Sets a new value for the object associated to the key.
        savedStateHandle.set(USER_KEY, userId)
    }

    fun getCurrentUser(): String {
        // Gets the current value of the user id from the saved state handle
        return savedStateHandle.get(USER_KEY)?. ""
    }
}
```

Usually you will use LiveData in your ViewModel. For that you can use the [SavedStateHandle.getLiveData\(\)](#) method. Here's an example of replacing getCurrentUser with a LiveData, which allows for observation:

```
// getLiveData gets MutableLiveData associated with a key.
// When the value associated with the key updates, the MutableLiveData does as well.
private val _userId : MutableLiveData<String> = savedStateHandle.getLiveData(USER_KEY)

// Only expose a immutable LiveData
val userId : LiveData<String> = _userId
```

SavedStateHandle also has other methods you might expect when interacting with a key-value map:

- [contains\(String key\)](#) - Checks if there is a value for the given key.
- [remove\(String key\)](#) - Removes the value for the given key.
- [keys\(\)](#) - Returns all keys contained within the SavedStateHandle.

For supported types please refer to <https://developer.android.com/topic/libraries/architecture/viewmodel-savedstate#types>

Transform LiveData

You may want to make changes to the value stored in a LiveData object before dispatching it to the observers, or you may need to return a different LiveData instance based on the value of another one. The [Lifecycle](#) package provides the [Transformations](#) class which includes helper methods that support these

scenarios.

Transformations.map()

Applies a function on the value stored in the `LiveData` object, and propagates the result downstream.

Kotlin Java

```

val userLiveData: LiveData<User> = UserLiveData()
val userName: LiveData<String> = Transformations.map(userLiveData) {
    user -> "${user.name} ${user.lastName}"
}
                    
```

Transformations.switchMap()

Similar to `map()`, applies a function to the value stored in the `LiveData` object and unwraps and dispatches the result downstream. The function passed to `switchMap()` must return a `LiveData` object, as illustrated by the following example:

Kotlin Java

```

private fun getUser(id: String): LiveData<User> {
    ...
}
val userId: LiveData<String> = ...
val user = Transformations.switchMap(userId) { id -> getUser(id) }
                    
```

In both **map** and **switchMap** there is a **source** (or trigger) live data, and in both cases you want to **transform** it to another **live data**. Which one will you use - depends on the task that your transformation is doing.

Map() is conceptually identical to the use in RXJava, basically you are changing a parameter of LiveData in another one

SwitchMap() instead you are going to substitute the LiveData itself with another one! Typical case is when you retrieve some data from a Repository for instance and to "eliminate" the previous LiveData (to garbage collect, to make it more efficient the memory usually) you pass a **new** LiveData that execute the same action(getting a query for instance)

To understand that difference Let's take an example, there is a LiveData which emits a string and we want to display that string in capital letters:

With map (in activity or fragment)

```
Transformations.map(stringsLiveData, String::toUpperCase)
    .observe(this, textView::setText);
```

the function passed to the map returns a string only, but it's the Transformation#map which ultimately returns a LiveData.

With SwitchMap (also in activity or fragment)

```
Transformations.switchMap(stringsLiveData, this::getUpperCaseStringLiveData)
    .observe(this, textView::setText);

private LiveData<String> getUpperCaseStringLiveData(String str) {
    MutableLiveData<String> liveData = new MutableLiveData<>();
    liveData.setValue(str.toUpperCase());
    return liveData;
}
```

If you see Transformations#switchMap has actually switched the LiveData. So, again as per the documentation **The function passed to switchMap() must return a LiveData object.**

So, in case of map it is the **source** LiveData you are transforming and in case of switchMap the passed LiveData will act as a **trigger** on which it will switch to another LiveData after unwrapping and dispatching the result downstream.

Transformation are useful because they are computed lazily (meaning that they are not calculated unless someone is observing their returned LiveData) that's why they goes well with the observer's lifecycle without any additional configuration.

They are very useful in case where a change in one object should return another one. For example if we have a UI component that gets and address and return postal code, then the we must register to a LiveData returned from our repository. We can do it like this:

```
class MyViewModel(private val repository: PostalCodeRepository) : ViewModel() {

    private fun getPostalCode(address: String): LiveData<String> {
        // DON'T DO THIS
        return repository.getPostCode(address)
    }
}
```

Which is not a good idea for two reasons the first is that each time the activity or fragment is recreated and we are doing a new database fetch because we don't store the old value but rather fetching it all over again and the second one is that each time he calls this function he is actually registering a new Live data which is costly.

What we should be doing in that case is:


```
class MyViewModel(private val repository: PostalCodeRepository) : ViewModel() {
    private val addressInput = MutableLiveData<String>()
    val postalCode: LiveData<String> = Transformations.switchMap(addressInput) {
        address -> repository.getPostCode(address) }

    private fun setInput(address: String) {
        addressInput.value = address
    }
}
```

In this case, the postalCode field is defined as a transformation of the addressInput. As long as your app has an active observer associated with the postalCode field, the field's value is recalculated and retrieved whenever addressInput changes and that's it, no extra calculations are done.

<https://medium.com/androiddevelopers/viewmodels-with-saved-state-jetpack-navigation-data-binding-and-coroutines-df476b78144e>

ViewModel NavGraph Integration

Before we saw that we can share information between fragments and their hosting activity using the shared View Model that we can access from all fragments:

```
// Any fragment's onCreate or onActivityCreated
// This ktx requires at least androidx.fragment:fragment-ktx:1.1.0
val sharedViewModel: ActivityViewModel by activityViewModels()
```

But what can we do if we want a shared view Model by some of the fragments and not all of them, While they all share the same Activity?

The solution is to this is to create a **nested navigation graph and share a view model to that graph**

The new Navigation API introduces ViewModels associated to a Navigation Graph. In practice, this means you can take a collection of associated destinations, such as an onboarding flow, a login flow, or a checkout flow; put them into a [nested navigation graph](#); and enable shared data just between those screens.

To create a nested navigation graph, you can select your screens, right click, and select **Move to Nested Graph → New Graph**:

In the XML view, note the **id** of the nested navigation graph, in this case checkout_graph:

```
<navigation app:startDestination="@id/homeFragment" ...>
  <fragment android:id="@+id/homeFragment" .../>
  <fragment android:id="@+id/productListFragment" .../>
  <fragment android:id="@+id/productFragment" .../>
  <fragment android:id="@+id/bargainFragment" .../>

  <navigation
    android:id="@+id/checkout_graph"
    app:startDestination="@id/cartFragment">

    <fragment android:id="@+id/orderSummaryFragment".../>
    <fragment android:id="@+id/addressFragment" .../>
    <fragment android:id="@+id/paymentFragment" .../>
    <fragment android:id="@+id/cartFragment" .../>

  </navigation>

</navigation>
```

Once you've done this, you get the ViewModel using by navGraphViewModels:

```
val viewModel: CheckoutViewModel by navGraphViewModels(R.id.checkout_graph)
```

But don't forget to add the KTX dependency:

```
implementation 'androidx.navigation:navigation-fragment-ktx:2.4.2'
```

To check it in our project simply a create a ViewModel with a single int property

```
class NavGraphViewModel : ViewModel() {
  val x = 9
}
```

In each of the fragments get a reference to it by using the navGraphViewModel and supply it with your root navigation graph id and simply Toast the value stored in your view model

```
class SignUpFragment : Fragment() {  
  
    private var _binding : SignUpFragmentBinding? = null  
    private val binding get() = _binding!!  
  
    private val viewModel : NavGraphViewModel by navGraphViewModels(R.id.my_nav)  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? {  
  
        Toast.makeText(requireContext(), text: "${viewModel.x}", Toast.LENGTH_SHORT).show()  
    }  
}
```

```
class FinishFragment() : Fragment() {  
  
    private var _binding : FinishFragmentBinding? = null  
    private val binding get() = _binding!!  
  
    private val viewModel : NavGraphViewModel by navGraphViewModels(R.id.my_nav)  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? {  
  
        Toast.makeText(requireContext(), text: "${viewModel.x}", Toast.LENGTH_SHORT).show()  
    }  
}
```

ViewModelProvider Factory

Sometimes we want to initiate our view model and pass parameters to his constructor so that we can use it's init function for data fetching request for example.

We will see that Hilt can inject necessary components such as a repository that handles network and database requests. This is the preferred way, And it will be show in the Hilt Model.

However, if you don't want to use Hilt or simply Dagger you can inherit from the NewInstanceFactory which implement the ViewModelProvider.Factory interface and override it's create() function. Pass your arguments to your factory implementation and in it call you view model constructor with the parameters and return your constructor initiated view model. Pass your implementation to the ViemodelProvider constructor which also gets a factory method besides the lifecycle owner and That's it!

Let's say we want to pass this view model a string so he can use it in his init function to initiate a database fetch

```
class ParamConstructedViewModel(var str:String): ViewModel() {
    init {
        //TODO: Use 'str' to init process when VM is created. i.e. Get data request.
        str = "$str dsdfs"
    }
}
```

Now we need to create his Factory implementation (we pass the factory params to our view model constructor):

```
// Override ViewModelProvider.NewInstanceFactory to create the ViewModel (VM).
class ParamConstructedModelFactory(private var str: String)
    : ViewModelProvider.NewInstanceFactory() {
    override fun <T : ViewModel?>
        create(modelClass: Class<T>): T = ParamConstructedViewModel(str) as T
}
```

And in the fragment or activity we can do this:

```
private lateinit var viewModel : ParamConstructedViewModel

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    viewModel = ViewModelProvider( owner: this,
        ParamConstructedModelFactory( str: "another string"))[ParamConstructedViewModel::class.java]

    Toast.makeText( context: this,viewModel.str,Toast.LENGTH_SHORT).show()
```

Or use the KTX lazy delegate:

```
class MainActivity : AppCompatActivity() {
    private val viewModel : ParamConstructedViewModel by viewModels {
        ParamConstructedModelFactory( str: "some string")
    }
}
```

Room

Download the full App Created in this Guide:

<https://drive.google.com/file/d/1tVbM7C7klt0rw3FcmelwZ1GON2GxJiGR/view?usp=sharing>

Let's say we need to persist the data that added by the user. So we want a persistence library like *Room*. It is an Object-Mapping library that provides an abstraction layer over SQLite that doesn't try to hide SQLite but rather embrace it.

There are some reasons why we use room for data persistence:

- It's based on SQLite so we can write SQL queries. Don't forget that Android supports SQLite as a proven technology from the day one.
- With Room we can have observability, because it can return LiveData objects and does the object mapping for you.
- Room creates the database schema using your entity definitions and does the sql operations like *insert*, *update* and *delete* using annotation processing, resulting in lesser boilerplate code.
- Room speaks SQL. So it knows whether you made a typo or did something wrong in queries at compile time.
- You can create abstract suspended functions and let room create all the logic for you.

Other SQLite Android libraries:

- 1. Sugar ORM** – This is an object relational mapper that wrap SQLite database. It map sqlite table to a java plain object.
- 2. Realm Database** – It provides offline-first functionality & data persistence through an easy-to-use API.
- 3. SQLBrite** – A lightweight wrapper around SQLiteOpenHelper which introduces reactive stream semantics to SQL operations

Primary components

There are three major components in Room:

- **Data entities** that represent tables in your app's database.
- **Data access objects (DAOs)** that provide methods that your app can use to query, update, insert, and delete data in the database.
- The **database class** that holds the database and serves as the main access point for the underlying connection to your app's persisted data.

First Step - Add the latest room library

Visit [this page](#) and import the room library to your app Gradle project file:

```
def room_version = "2.4.2"

implementation "androidx.room:room-runtime:$room_version"
kapt "androidx.room:room-compiler:$room_version"
```

Important Note:

In the app Gradle Add the plugin :
id 'kotlin-kapt'

Please note we replaced the room annotationProcessor to kapt - Kapt is the Kotlin Annotation Processing Tool. If you want to be able to reference generated code from Kotlin, you need to use kapt.

Second Step - Define your Entities

Use the **@Entity** annotation to define a new entity - this you will do for the basic Kotlin or Java class you want to save in your database - you can optionally give the table a name using (tableName = "[table name]") the default table name is the class name.

Define your primary key using @PrimaryKey next to the property. That will serve as your primary key. If you don't have a unique key to your objects like emails you can set it to be auto generated use (autoGenerate = true) next to it.

If you want a different column name in the data base from the property name use @ColumnInfo(name = ["Your name"])

It's recommended you always use the @ColumnInfo annotation as it gives you more flexibility to rename the members without having to change the database column names. Changing the column names leads to a change in the database schema and therefore you need to implement a migration or specific instructions not to implement migration.

For example:

```
@Entity(tableName = "items_table")
data class Item(
    @ColumnInfo(name = "content")
    val content:String,

    @ColumnInfo(name = "description")
    val description:String,

    @ColumnInfo(name = "image")
    val image:String?)
{

    @PrimaryKey(autoGenerate = true)
    var id : Int = 0
}
```

Third step - define you Dao classes

Dao classes will allow you to abstract the database communication in a more logical layer which will be much easier to mock in tests (compared to running direct sql queries). It also automatically does the conversion from Cursor to your application classes so you don't need to deal with lower level database APIs for most of your data access.

Room also verifies all of your queries in **Dao** classes while the application is being compiled so that if there is a problem in one of the queries, you will be notified instantly while you are writing it.

The class marked with @Dao should either be an interface or an abstract class. At compile time, Room will generate an implementation of this class when it is referenced by a Database.

An abstract @Dao class can optionally have a constructor that takes a Database as its only parameter.

It is recommended to have multiple Dao classes in your codebase depending on the tables they interact.

```
@Dao
interface ItemsDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun addItem(item: Item)

    @Delete
    fun deleteItem(vararg item: Item)

    @Update
    fun update(item: Item)

    @Query(value = "SELECT * from items_table ORDER BY content ASC")
    fun getItems() : LiveData<List<Item>>

    @Query(value = "SELECT * from items_table WHERE content LIKE :title")
    fun getItem(title:String) : Item
}
```

onConflict annotation parameter signifies what to do if a conflict happens on insertion. It can take the following values:

- **OnConflictStrategy.REPLACE** : To replace the old data and continue the transaction.
- **OnConflictStrategy.ABORT** : To abort the transaction. The transaction is rolled back.
- **OnConflictStrategy.NONE** : To ignore the conflict.

Third step - create you database

Now create the AppDatabase class to hold the database. AppDatabase defines the database configuration and serves as the app's main access point to the persisted data. The database class must satisfy the following conditions:

- The class must be annotated with a **@Database** annotation that includes an **entities** array that lists all of the data entities associated with the database.
- The class must be an abstract class that extends **RoomDatabase**.
- For each DAO class that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class.

In order to make sure we don't have multiple database instances open at the same time we define a RoomDatabase instance in the companion object of our class. We need the application context to initialize the database. So the best way to handle this is to add a getDatabase function that receives the context

and builds the database.

We'll define an abstract method that returns the ItemsDao. Everything is abstract because Room is the one that generates the implementation for us.

exportSchema

You can set annotation processor argument to tell Room to export the schema into a folder. Even though it is not mandatory, it is a good practice to have version history in your codebase and you should commit that file into your version control system (but don't ship it with your app!).

So if you don't need to check the schema and you want to get rid of the warning, just add `exportSchema = false` to your RoomDatabase

@Volatile - Volatile means, it will not be stored in the local cache. Meaning: writes to this field are immediately made visible to other threads.

```
@Database(entities = arrayOf(Item::class),version = 1, exportSchema = false)
abstract class ItemsDatabase : RoomDatabase() {

    abstract fun itemsDao() : ItemsDao

    companion object {

        @Volatile
        private var instance:ItemsDatabase? = null

        fun getDatabase(context:Context) : ItemsDatabase? {

            if(instance == null) {
                synchronized(ItemsDatabase::class.java) {
                    instance = Room.databaseBuilder(context.applicationContext,
                        ItemsDatabase::class.java, name: "items_database").build()
                }
            }
            return instance
        }
    }
}
```

```
//or in more Kotlin way
fun getDatabase(context:Context) = instance ?: synchronized( lock: this) {
    Room.databaseBuilder(context.applicationContext,ItemsDatabase::class.java, name: "items_database")
        .allowMainThreadQueries().build().also { instance = it }
}
```

Forth step(optional)

create your helper class the gives a single access point to your database
You can think of repository as the single access point for getting the data. The

class will include all the functions from which we can get all the data.

```
class ItemsRepository(application: Application) {

    private var itemsDao:ItemsDao?

    init {
        val db = ItemsDatabase.getDatabase(application)
        itemsDao = db?.itemsDao()
    }

    fun getItems() = itemsDao?.getItems()

    fun addItem(item: Item) {
        itemsDao?.addItem(item)
    }
}
```

Please note that while the fetching that return LiveData is done on a background thread automatically, adding the item is done on the application main thread! until we use coroutines you can use this but be sure to add the .allowMainThreadQueries() to your ROOM database builder. If you don't do this the app will crash and you will get an error message "Cannot access database on the main".

```
instance = Room.databaseBuilder(context.applicationContext,
    [ItemsDatabase::class.java, name: "items_database").allowMainThreadQueries().build()
```

Later we will solve this by adding Coroutines to our project.

Fifth step

Update your view model

Here we need to extend the Android ViewModel because we need the application instance to give to our database

```
class ItemViewModel(application: Application) : AndroidViewModel(application) {  
  
    private var repository = ItemsRepository(application)  
  
    fun getItems() = repository.getItems()  
  
    fun addItem(item : Item) {  
        repository.addItem(item)  
    }  
}
```

And That's it you have integrated ROOM. Go ahead and check your implementation by creating the full project

Let's add our ItemViewModel

In our case we want to share one ViewModel for the whole activity because it makes sense. We have three Fragments all need to access the same information: one shows the list, one adds an item to it and one shows a single item from the list. The ViewModel will hold all the items as a LiveData property and the chosen item also as LiveData, it will have functions to add an item, delete an item, delete all items and set the chosen item. Please note that we will inherit from the **AndroidViewModel** because we need the Context in order to create the repository private instance who will serve as a single access point for our data.

Our final View Model will look like this (please note that while the chosen item must be a mutable live data we only expose it as Live Data in order to keep our data persistent):

```
class ItemsViewModel(application: Application) : AndroidViewModel(application){

    private val repository = ItemRepository(application)

    val items : LiveData<List<Item>>? = repository.getItems()

    private val _chosenItem = MutableLiveData<Item>()
    val chosenItem : LiveData<Item> get() = _chosenItem

    fun setItem(item:Item) {
        _chosenItem.value = item
    }

    fun addItem(item: Item) {
        repository.addItem(item)
    }

    fun deleteItem(item:Item) {
        repository.deleteItem(item)
    }

    fun deleteAll() {
        repository.deleteAll()
    }
}
```

Note that we also added a delete all function to the repository and to the Dao:

```
fun deleteAll() {
    itemDao?.deleteAll()
}
```

```
@Query( value: "DELETE FROM items")
fun deleteAll()
```

Now let's go back to the UI - To all fragment get the view Model bounded to the activity scope and because all of them lives in the same activity there will be one instance of it that will be shared by all of them. So add this line to the top of **each fragment**:

```
private val viewModel : ItemsViewModel by activityViewModels()
```

Go to the **AllItemsFragment** In the **onViewCreated** get the viewModel's items LiveData and observe it. In the callback which is called with the updated list of items pass it to the adapter and implement the callback functions (remove the code with the ItemManager and replace it with this):

```
viewModel.items?.observe(viewLifecycleOwner) { it: List<Item>!

    binding.recycler.adapter = ItemAdapter(it, object : ItemAdapter.ItemListener {

        override fun onItemClick(index: Int) {

            Toast.makeText(requireContext(),
                text: "${it[index]}", Toast.LENGTH_SHORT).show()

        }

        override fun onItemLongClicked(index: Int) {
            viewModel.setItem(it[index])
            findNavController().navigate(R.id.action_allItemsFragment_to_detailItemFragment)
        }

    })
    binding.recycler.layoutManager = LinearLayoutManager(requireContext())
}
}
```

Before finishing the AllItemsFragment go ahead to your adapter and add a function that will return an Item according to the position, because in the fragment upon swiping we get the position but we need to pass an Item to the viewModel delete function. So we will add a function to get an item according to its position(in ItemAdapter):

```
fun itemAt(position: Int) = items[position]
```

Now on the onSwipe of the ItemTouchHelper use this function:

```
override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int) {
    val item = (binding.recycler.adapter as ItemAdapter).itemAt(viewHolder.adapterPosition)
    viewModel.deleteItem(item)
    // ItemManager.remove(viewHolder.adapterPosition)
    // binding.recycler.adapter!!.notifyItemRemoved(viewHolder.adapterPosition)
}
}).attachToRecyclerView(binding.recycler)
```

In the **AddItemFragment** remove the ItemManger access and replace it with the viewModel call (remember you added it as a property before):

```
val item = Item(binding.itemTitle.text.toString(),
    binding.itemDescription.text.toString(), imageUrl.toString())
// ItemManager.add(item)

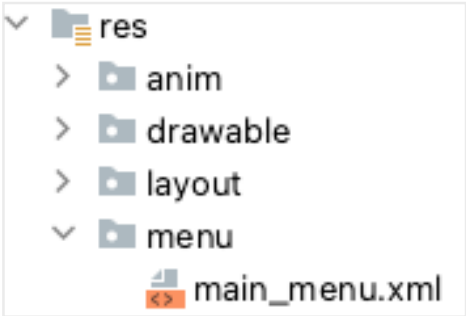
viewModel.addItem(item)
```

And in the DetailItemFragment after the view had been created observe your chosen item live data and update your UI (what was previously located in the arguments let scope):

```
viewModel.chosenItem.observe(viewLifecycleOwner) { it: Item!
    binding.itemTitle.text = it.title
    binding.itemDesc.text = it.description
    Glide.with(requireContext()).load(it.photo).circleCrop()
        .into(binding.itemImage)
}
```

Adding a Menu

In order to add an action/option menu to the top of the AllItemsFragment we create the following xml file under the resource menu folder:



```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item android:id="@+id/action_delete"
        android:title="Delete all"
        android:icon="@drawable/ic_baseline_delete_outline_24"
        app:showAsAction="always"/>
</menu>
```

This menu item contains an id, a vector asset we added before (just right click on the res folder choose new and then choose vector asset, in the new window choose a trash bin from the clipart and click finish), and a showAsAction attribute set to always means it will be shown all the time on the menu and not under the three dots (try giving different values).

In the AllItemsFragment where the menu is shown override these two functions:

```

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.main_menu, menu)
    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    if(item.itemId == R.id.action_delete) {
        val builder = AlertDialog.Builder(requireContext())
        builder.setTitle("Confirm delete")
        .setMessage("Are you sure you want to delete all items?")
        .setPositiveButton(text: "Yes")
        { p0, p1 ->
            viewModel.deleteAll()
            Toast.makeText(requireContext(), text: "Items deleted", Toast.LENGTH_SHORT).show()
        }.show()
    }
    return super.onOptionsItemSelected(item)
}

```

The first one will be called by the system when its time to create the option menu and we will use the given menu inflater to inflate our own xml menu file we just dud to the empty menu the os gives us.

The second function is called upon pressing the action menu items and after presenting a confirmation dialog we will use our viewModel deleteAll function.

Please note that in order for the fragment to show the Menu you must add this line in the onCreateView of the AllItemsFragment function:

```
setHasOptionsMenu(true)
```

this is only needed when presenting the menu in a fragment and not in the activity - if we would have presented the menu in the activity it would have existed throughout all go the fragments and we don't want that.

And that's it the project is finished run and test your app.

Please note that we still allow queries too run on the application main thread:

```
Room.databaseBuilder(context.applicationContext, ItemDataBase::class.java, name: "items_db")
    .allowMainThreadQueries().build()
```

Try to remove this line and test your app.

Please note the the getItems works just fine because it returns LiveData and LiveData by default is doing all of its work on the Dispatchers.IO group of threads which are background thread ads and not on the main thread but try to add an item and see what happens... YES the app crashed and ion the logical you can't find the following message:

```
java.lang.IllegalStateException: Cannot access database on the main thread since it may potentially lock the UI for a long period of time.
```

And that's why we need to study Coroutines (Come back to this tutorial afterwards).

Lets improve our background work(After the co-routine chapter):

The first solution is for repository to implement CoroutineScope and override CoroutineContext to operate in IO Thread.

```
class ItemsRepository(application: Application) : CoroutineScope {  
  
    override val coroutineContext: CoroutineContext  
        get() = Dispatchers.IO  
  
    private var itemsDao:ItemsDao?  
  
    init {  
        val db = ItemsDatabase.getDatabase(application)  
        itemsDao = db?.itemsDao()  
    }  
  
    fun getItems() = itemsDao?.getItems()  
  
    fun addItem(item: Item) {  
        launch { this: CoroutineScope  
            itemsDao?.addItem(item)  
        }  
    }  
}
```

Now you can remove the `allowMainThreadQueries()` from your database instance initialization and go ahead and run your app and try adding an item... No crash!

Try deleting and it crashes again, so do the same for the delete functions they are not returning LiveData:


```
fun deleteItem(item: Item) {  
    launch { this: CoroutineScope  
        itemDao?.deleteItem(item)  
    }  
}  
  
fun getItem(id:Int) = itemDao?.getItem(id)  
  
fun deleteAll() {  
    launch { this: CoroutineScope  
        itemDao?.deleteAll()  
    }  
}
```

The problem with this solution is that it is not subject to the principle of *structured concurrency* meaning there is no actual scope confined to any lifecycle for these coroutines. So although it works we can make it better.

The best option is use the ROOM coroutine KTX extensions and our view model scope.

Room nows comes with coroutine support. DAO methods can now be marked as suspended to ensure that they are not executed on the main thread. We can make the Dao addItem() function to be suspended and then Room will generate the code using coroutines by himself but we must call it from another suspended function or a coroutine context so we will use the viewModel scope to execute it.

Just add the following dependency to your app grade file:
implementation("androidx.room:room-ktx:\$room_version")

And make the DAO insert, update and delete functions suspended:

```

@Dao
interface ItemDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun addItem(item:Item)

    @Delete
    suspend fun deleteItem(vararg items:Item)

    @Update
    suspend fun updateItem(item: Item)

    @Query(value: "DELETE FROM items")
    suspend fun deleteAll()

    @Query(value: "SELECT * FROM items ORDER BY id")
    fun getItem() : LiveData<List<Item>>

    @Query(value: "SELECT * FROM items WHERE id = ?")
    fun getItem(id:Int) : Item
    
```

That's it under the hood ROOM automatically replaces this auto implemented synchronous code:

```

@Override
public void insertUserSync(final User user) {
    __db.beginTransaction();
    try {
        __insertionAdapterOfUser.insert(user);
        __db.setTransactionSuccessful();
    } finally {
        __db.endTransaction();
    }
}
    
```

With this:

```
@Override
public Object insertUserSuspend(final User user,
    final Continuation<? super Unit> p1) {
    return CoroutinesRoom.execute(__db, new Callable<Unit>() {
        @Override
        public Unit call() throws Exception {
            __db.beginTransaction();
            try {
                __insertionAdapterOfUser.insert(user);
                __db.setTransactionSuccessful();
                return kotlin.Unit.INSTANCE;
            } finally {
                __db.endTransaction();
            }
        }
    }, p1);
}
```

The generated code ensures that the insert happens off of the UI thread. In our suspend function implementation, the same logic from the synchronous insert method is wrapped in a `Callable`. Room calls the `CoroutinesRoom.execute` suspend function, which switches to a background dispatcher, depending on whether the database is opened and we are in a transaction or not. If we check the [CoroutinesRoom.execute\(\) implementation](#), we see that Room moves `callable.call()` to a different CoroutineContext. This is derived from the executors you provide when building your database or by default will use the Architecture Components IO Executor.

So the actual changes in our code is making the Dao functions suspended and because it is called from the repository functions we should make them also suspended and execute it from the `viewModelScope`:

```
suspend fun addItem(item:Item) {  
    //launch {  
        itemDao?.addItem(item)  
    //}  
}  
  
suspend fun deleteItem(item: Item) {  
    // launch {  
        itemDao?.deleteItem(item)  
    // }  
}  
  
fun getItem(id:Int) = itemDao?.getItem(id)  
  
suspend fun deleteAll() {  
    // launch {  
        itemDao?.deleteAll()  
    //}  
}
```

```
fun addItem(item: Item) {  
    viewModelScope.launch { this: CoroutineScope  
        repository.addItem(item)  
    }  
}  
  
fun deleteItem(item:Item) {  
    viewModelScope.launch { this: CoroutineScope  
        repository.deleteItem(item)  
    }  
}  
  
fun deleteAll() {  
    viewModelScope.launch { this: CoroutineScope  
        repository.deleteAll()  
    }  
}
```

viewModelScope is a Kotlin extension property on the ViewModel class. It is a CoroutineScope that is cancelled once the ViewModel is destroyed (when [onCleared\(\)](#) is called). Thus when you're using a ViewModel, you can start all of your coroutines using this scope.

Please note that [@Transaction](#) methods can also be suspended and they can call other suspended DAO functions:

```
@Dao
abstract class UsersDao {

    @Transaction
    open suspend fun setLoggedInUser(loggedInUser: User) {
        deleteUser(loggedInUser)
        insertUser(loggedInUser)
    }

    @Query("DELETE FROM users")
    abstract fun deleteUser(user: User)

    @Insert
    abstract suspend fun insertUser(user: User)
}
```

Room offers a lot of functionality and flexibility than what we've covered — you can define how Room should handle database conflicts, you can store types that otherwise, natively with SQLite can't be stored, [like Date](#), by creating TypeConverters, you can implement complex queries, using JOIN and other SQL functionality, [create database views](#), pre-populate your database or trigger certain database actions whenever the database is created or opened.

For more reading please refer to

<https://developer.android.com/training/data-storage/room>

Coroutines Kotlin

Download the Full App created in this Guide:

<https://drive.google.com/file/d/1PDhzzYXb4IRXYNuniU2YMpQflv07JAKZ/view?usp=sharing>

Download the Architecture Project with Coroutines support

https://drive.google.com/file/d/1azpucisy6VxHNAPIVg2EecD_MqAxn4L7/view?usp=sharing

Coroutines is Google's recommended solution for asynchronous programming on Android.

Coroutines are:

- **Lightweight:** You can run many coroutines on a single thread due to support for *suspension*, which doesn't block the thread where the coroutine is running. Suspending saves memory over blocking while supporting many concurrent operations.
- **Fewer memory leaks:** Use *structured concurrency* to run operations within a scope.
- **Built-in cancellation support:** *Cancellation* is propagated automatically through the running coroutine hierarchy.
- **Jetpack integration:** Many Jetpack libraries include *extensions* that provide full coroutines support. Some libraries also provide their own *coroutine scope* that you can use for structured concurrency.

Co - cooperate

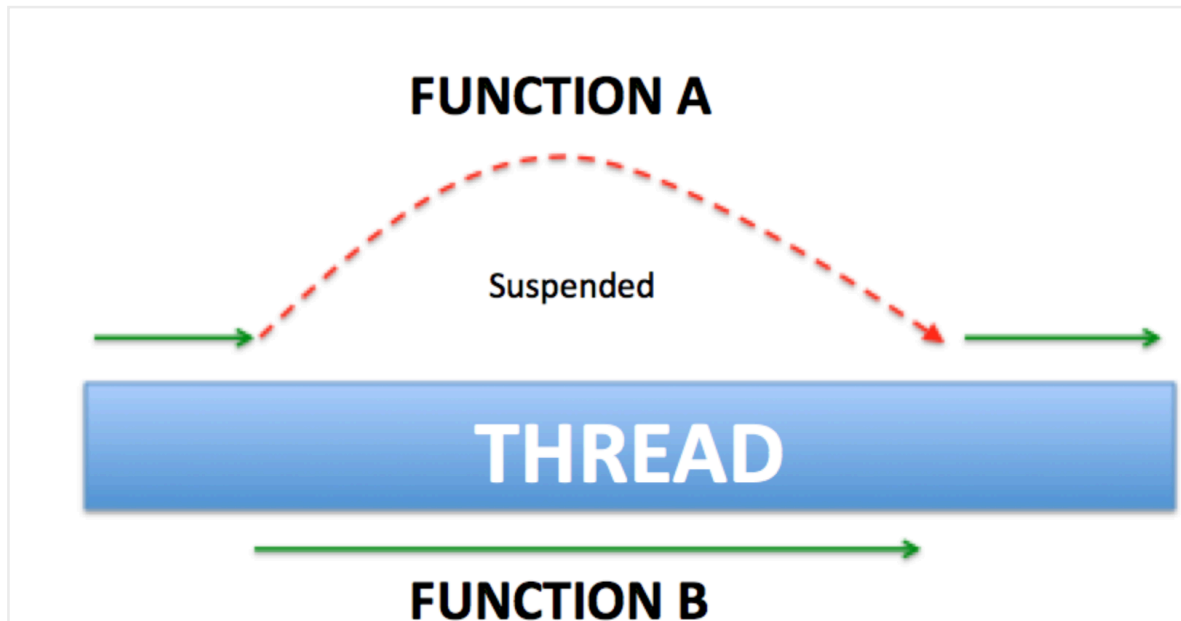
Routines - functions

Timing functions

One can think of a coroutine as a light-weight thread. Like threads, coroutines can run in parallel, wait for each other and communicate. The biggest difference is that coroutines are very cheap, almost free: we can create thousands of them, and pay very little in terms of performance. True threads, on the other hand, are expensive to start and keep around. A thousand threads can be a serious challenge for a modern machine.

Suspended functions are at the center of everything in coroutines. A suspended function is simply a function that can be paused and resumed at a later time. They can execute a long running operation and wait for it to complete without blocking. You can even stop or suspend your function while you are waiting for a callback and continue it when you get the result. This is why You can run many coroutines on a single thread. Suspension doesn't block the thread where the coroutine is running. We save allot of

memory by not blocking the thread.



The exact definition of Coroutines: A framework to manage concurrency in a more performant and simple way with its lightweight thread which is written on top of the actual threading framework to get the most out of it by taking the advantage of cooperative nature of functions.

ViewModel, LiveData and Lifecycle includes a set of KTX extensions that work directly with coroutines. We will see later on.

First you need to import the latest Kotlin coroutines to your android studio project. You can find the latest version here. Follow the Gradle instructions. Add both the core and the android libraries :

<https://github.com/Kotlin/kotlinx.coroutines>

```
implementation 'org.jetbrains.kotlin:kotlinx-coroutines-android:1.6.0'
implementation 'org.jetbrains.kotlin:kotlinx-coroutines-core:1.6.0'
```

Then create the following activity that suppose to fetch a user from the database (fetching illustrated here by sleep - sometimes working and sleeping is exactly the same:)) and updating it's details in Text View(make sure your default activity_main.xml file has a Text View whose id is text_view and use view binding.


```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        fetchAndShowUser()
    }

    private fun fetchAndShowUser() {
        val user = fetchUser()
        showUser(user)
    }

    private fun fetchUser() : User {
        Thread.sleep( millis: 3000) //no checked exceptions on kotlin
        return User( name: "Moshe", email: "moshe@moshe.com")
    }

    private fun showUser(user:User) {
        binding.textView.text = user.toString()
    }
}

data class User(val name:String, val email:String)
```

If we execute it like this it would result in a very poor user experience - a stuck app for 3 secs!

We have to define a Coroutine scope.

Coroutine scope promotes [structured concurrency](#), whereby you can launch multiple coroutines in the same scope and cancel the scope (which in turn cancels all the coroutines within that scope).

if we are not already in a coroutine scope we can use the **GlobalScope** which is the scope of all the app. As long as the app is alive all our coroutines can run in this scope (there are more scopes and working with this scope is not recommended)

A global CoroutineScope not bound to any job. Global scope is used to launch top-level coroutines which are operating on the whole application lifetime and

are not cancelled prematurely.

Active coroutines launched in GlobalScope do not keep the process alive. They are like daemon threads.

This is a delicate API. It is easy to accidentally create resource or memory leaks when GlobalScope is used. A coroutine launched in GlobalScope is not subject to the principle of structured concurrency, so if it hangs or gets delayed due to a problem (e.g. due to a slow network), it will stay working and consuming resources until the app finishes.

From the GlobalScope we can call **launch()** that can execute our coroutines and return a job object that can be started if the coroutine start is lazy (by default all coroutines created with this function are executed immediately) or cancelled later on, or **async()** which is almost the same except it returns a Deferred object containing the Coroutine result. The coroutine created with this function is cancelled if the deferred object is cancelled.

Both function need a CoroutineContext. The coroutine Context has a default value.

The coroutine context includes a *coroutine dispatcher* that determines what thread or threads the corresponding coroutine uses for its execution.

The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.

CoroutineDispatcher tells the coroutine builder (in our case launch{} or async{}) as to which pool of threads is to be used. There are a few predefined Dispatchers available.

- **Dispatchers.Default:** CPU-intensive work, such as sorting large lists, doing complex calculations and similar. A shared pool of threads on the JVM backs it.
- **Dispatchers.IO:** networking or reading and writing from files. In short – any input and output, as the name states
- **Dispatchers.Main:** mandatory dispatcher for performing UI-related events in Android's main or UI thread.
- **Dispatchers.Unconfined** - A coroutine dispatcher that is not confined to any specific thread. The unconfined dispatcher is appropriate for coroutines which neither consume CPU time nor update any shared data (like UI) confined to a specific thread. From the Kotlin docs: The unconfined dispatcher is an advanced mechanism that can be helpful in certain corner cases where dispatching of a coroutine for its execution later is not needed or produces undesirable side-effects, because some operation in a coroutine must be performed right away. The unconfined dispatcher should not be used in general code.

When **launch {}** is used without parameters, it inherits the context (and thus dispatcher) from the **CoroutineScope** it is being launched from. In this case of

the Global Scope its default context is the the Dispatcher.Default. We can specify Any other Context we want to the GloablScope and run the launch or async result on the Dispatcher.IO like that:

```
GlobalScope.launch(Dispatchers.IO) {}
```

Try commenting out the existing code and run the following (watch the output):

```
GlobalScope.launch {
    println("Default    : I'm working in thread ${Thread.currentThread().name}")
}
GlobalScope.launch(Dispatchers.IO) {
    println("IO        : I'm working in thread ${Thread.currentThread().name}")
}
GlobalScope.launch(Dispatchers.Main) {
    println("Main       : I'm working in thread $
{Thread.currentThread().name}")
}
GlobalScope.launch(newSingleThreadContext("MyOwnThread")) { // will get its
own new thread
    println("newSingleThreadContext: I'm working in thread $
{Thread.currentThread().name}")
}
```

So to shorten our definitions: Each coroutine is a Job, a job must run in a scope for efficient memory management and must receive a Context which include the Dispatcher - which threads the coroutine will run on.

First step:

```
GlobalScope.launch(Dispatchers.IO) { this: CoroutineScope
    fetchAndShowUser()
}
```

This will cause the app to crash when updating ui from the background!

If we call **async** we get Deferred value that can be extract using **await()** - this is very similar to Future in Java

Please note the await() is a suspended function (it make sense cause it can't run before the function that it is working on will finish) that can only be called from another suspended function or a Coroutine context.

Second step:

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    GlobalScope.launch(Dispatchers.Main) { this: CoroutineScope
        fetchAndShowUser()
    }
}

private suspend fun fetchAndShowUser() {
    val user = fetchUser()
    showUser(user)
}

private suspend fun fetchUser() = GlobalScope.async(Dispatchers.IO) {
    Thread.sleep(millis: 3000) //no checked exceptions on kotlin
    User(name: "Moshe", email: "moshe@moshe.com")
}.await()

```

Another option instead of turning the fetch function to suspended is to call `await()` in the `fetchAndShowUser`

```

private suspend fun fetchAndShowUser() {
    val user = fetchUser()
    showUser(user.await())
}

```

Instead of calling from the `GlobalScope` and await for the result we can use the function `withContext` that creates a suspended function that runs in the given (or from the scope) `Context`.

Complete definition: Calls the specified suspending block with a given coroutine context, suspends until it completes, and returns the result. Before the KTX this is what we used.

suspend fun <T> withContext(context: CoroutineContext, block: suspend CoroutineScope.() -> T): T

```
private suspend fun fetchUser() = withContext(Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe", email: "moshe@moshe.com")
}
```

withContext like await() it is also a suspended function (it make sense since it must suspend the coroutine until the work is finished) so the fetchUser must also be suspended.

Let's see what are the consequences of using withContext:

Add another Text View to your xml file and change both ids to text_view_one and text_view_two

For example this code will take 6 secs - First the first user is fetched and only then does the fetching of the second user starts (you can change the Thread.sleep() to the suspended function delay())

```
private suspend fun fetchAndShowUser() {
    val user1 = fetchUserOne()
    Log.d(TAG, msg: "user one fetched")

    val user2 = fetchUserTwo()
    Log.d(TAG, msg: "user two fetched")

    showUser(user1,user2)
}

private fun showUser(user1:User, user2:User) {
    binding.textViewOne.text = user1.toString()
    binding.textViewTwo.text = user2.toString()
}

private suspend fun fetchUserOne() = withContext(Dispatchers.IO) { this: Co
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe", email: "moshe@moshe.com")
}

private suspend fun fetchUserTwo() = withContext(Dispatchers.IO) { this: Co
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Misha", email: "misha@misha.com")
}
```

Which is exactly the same as this:

```
private suspend fun fetchUserOne() = GlobalScope.async(Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe", email: "moshe@moshe.com")
}.await()
```

```
private suspend fun fetchUserTwo() = GlobalScope.async(Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Misha", email: "misha@misha.com")
}.await()
```

Our fetch and show user waits for each user to be fetched and they are fetched one after the other and not simultaneously.

But if we change our code to the following all fetching work will at the same time:

```
private suspend fun fetchAndShowUser() {
    val user1 = fetchUserOne()
    Log.d(TAG, msg: "user one fetched")

    val user2 = fetchUserTwo()
    Log.d(TAG, msg: "user two fetched")

    showUser(user1.await(),user2.await())
}

private fun showUser(user1:User, user2:User) {
    binding.textViewOne.text = user1.toString()
    binding.textViewTwo.text = user2.toString()
}

private fun fetchUserOne() = GlobalScope.async(Dispatchers.IO) { this
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe", email: "moshe@moshe.com")
}

private fun fetchUserTwo() = GlobalScope.async(Dispatchers.IO) { this
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Misha", email: "misha@misha.com")
}
```

And this is an advantage async has, we can't achieve with witchContext!

Coroutines are not a new concept, let alone invented by Kotlin. They've been around for decades and are popular in some other programming languages such as Go. What is important to note though is that the way they're implemented in Kotlin, most of the functionality is delegated to libraries. In fact, beyond the suspend keyword, no other keywords are added to the language. This is somewhat different from languages such as C# that have async and await as part of the syntax. With Kotlin, these are just library functions.

A big improvement to the above code is taking advantage of the fact that the Activity itself can serve as the Coroutine Context for creating new Coroutines!

Our activity needs to implement the CoroutineScope interface and override the get method that returns the Coroutine context needed for calling the launch and async functions that can run our Coroutines. This way we don't need to call launch or async on the global scope. We also don't need to specify the Coroutine Context and the Dispatchers since the get() method returns the Context

And thus our code can be altered to this:

```
class MainActivity : AppCompatActivity(), CoroutineScope {  
  
    private lateinit var binding: ActivityMainBinding  
    private val TAG = MainActivity::class.java.name  
    override val coroutineContext: CoroutineContext  
        get() = Dispatchers.Main  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        binding = ActivityMainBinding.inflate(layoutInflater)  
        launch { this: CoroutineScope  
            fetchAndShowUser()  
        }  
    }  
  
    private suspend fun fetchAndShowUser() {  
        val user1 = fetchUserOneAsync()  
        Log.d(TAG, msg: "user one fetched")  
        val user2 = fetchUserTwoAsync()  
        Log.d(TAG, msg: "user two fetched")  
        showUser(user1.await(), user2.await())  
    }  
  
    private fun showUser(user1: User, user2: User) {  
        binding.textViewOne.text = user1.toString()  
        binding.textViewTwo.text = user2.toString()  
    }  
  
    private fun fetchUserOneAsync() = async(Dispatchers.IO) { this: CoroutineScope  
        Thread.sleep( millis: 3000) //no checked exceptions on kotlin  
        User( name: "Moshe", email: "moshe@moshe.com")  
    }  
  
    private fun fetchUserTwoAsync() = async(Dispatchers.IO) { this: CoroutineScope  
        Thread.sleep( millis: 3000) //no checked exceptions on kotlin  
        User( name: "Misha", email: "misha@misha.com")  
    }  
}
```

Again, when no Dispatchers is given to the launch or async coroutines it runs on the general context they are in.

So we don't have to specify to launch which scope and on which dispatcher to run in since it gets them from the activity.

Again one can think that all the coroutines launched from that scope will be auto cancelled when the activity is destroyed. But this is not the case. Just add this to the on create function. This code will create a coroutine that unless cancelled will last for a very long time. We will close our activity and see if the exception is thrown when the activity is destroyed and whether the coroutines ends with it:


```

Launch { this: CoroutineScope
    try {
        delay(Long.MAX_VALUE)
    } catch (e: Exception) {
        Log.d(TAG, msg: "coroutine cancelled")
        // e will be a JobCancellationException if the activity is destroyed
    }
}

```

We kill the activity and the exception is not thrown! No one killed the coroutine when onDestroyed got called.

This can be done by bounding the scope to a specific Job. First lets understand what is a Job:

Job

A coroutine itself is represented by a Job. A Job is a handle to a coroutine. For every coroutine that you create (by launch or async - deferred is also a job - you can cancel it), it returns a Job instance that uniquely identifies the coroutine and manages its lifecycle. You can also pass a Job to a CoroutineScope to keep a handle on its lifecycle.

The coroutine scope is determined by an empty Job it create for himself if you won't pass any and using the + operator he add this job to his internal hash map and detains the scope lifecycle by it.

```

public fun CoroutineScope(context: CoroutineContext): CoroutineScope =
    ContextScope(if (context[Job] != null) context else context + Job())

internal class ContextScope(context: CoroutineContext) : CoroutineScope {
    override val coroutineContext: CoroutineContext = context
    // CoroutineScope is used intentionally for user-friendly representation
    override fun toString(): String = "CoroutineScope(coroutineContext=$coroutineContext)"
}

```

```
interface Deferred<out T> : Job
```

So we will create an empty job in the concrete and cancel is on the OnDestroy and add it to the get function like this:

```
private lateinit var job: Job

override val coroutineContext: CoroutineContext
    get() = job + Dispatchers.Main

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    job = Job()
}
```

```
override fun onDestroy() {
    super.onDestroy()
    job.cancel()
}
```

No test your code again. Kill the activity. Is the exception thrown? Oh, ya...

We can create a lazy task that can be saved for later execution and will only be executed when needed, when we call start on the job returned.

This job will wait for the start function in oppose to regular launch call with default CoroutineStart parameter of DEFAULT and it start immediately

```
launch { this: CoroutineScope
    fetchAndShowUser()
}

val job = launch (start = CoroutineStart.LAZY){ this: CoroutineScope
    println("Task preformed")
}

job.start()
```

Another Coroutines useful suspended functions is:

joinAll() - that waits for all coroutines to return

Job.**join()** - called on specific Coroutine we want to wait for it to finish

repeat() - for repeated actions

delay(mills) which is much better than Thread.sleep because the later blocks

the whole thread while the former stops only the specific Coroutine and the others running on the same thread are not stopped.

Now we will see there is much more elegant solution for bounding our coroutines to the activity lifecycle that comes with the KTX-Extensions kit.

More on Coroutines Scope

To avoid work leaks you should organize your coroutines by adding them to a [CoroutineScope](#), which is an object that keeps track of coroutines.

CoroutineScopes can be cancelled; and when you cancel a scope, they cancel all the associated coroutines. Above I'm using the [GlobalScope](#), which is, as the name implies, a CoroutineScope that is available globally. It's generally **not** good practice to use the GlobalScope for the same reasons it's generally not good to write globally accessible variables. So you'll need to either make a scope, or get access to one.

In Activity or Fragment you can use the `lifecycleScope`

In ViewModels, this is easy if you use [viewModelScope](#).

And in LiveData you can use the `livedataScope`

Add the following implementations (if needed) in your app Gradle file:

- For `viewModelScope`, use `androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1` or higher.
- For `lifecycleScope`, use `androidx.lifecycle:lifecycle-runtime-ktx:2.4.1` or higher.
- For `livedataScope`, use `androidx.lifecycle:lifecycle-livedata-ktx:2.4.1` or higher.

LifecycleScope

A `LifecycleScope` is defined for each `Lifecycle` object. Any coroutine launched in this scope is canceled when the Lifecycle is destroyed. You can access the `CoroutineScope` of the Lifecycle either via `lifecycle.coroutineScope` or `lifecycleOwner.lifecycleScope` properties. It is important to understand that the default Dispatchers of the `lifecycleScope` is the `Dispatchers.Main` meaning the main thread of the application.

In an activity or fragment see the `lifecycleScope` like this:

```
// launch in main/ui thread
// lifecycleOwner.lifecycleScope.launch { }
lifecycleScope.launch { // Dispatchers.Main
}

// launch in background thread
lifecycleScope.launch(Dispatchers.Default) {
}
```

Even though the CoroutineScope provides a proper way to cancel long-running operations automatically, you might have other cases where you want to suspend execution of a code block unless the Lifecycle is in a certain state. For example, to run a FragmentTransaction, you must wait until the Lifecycle is at least STARTED. For these cases, Lifecycle provides additional methods: lifecycle.whenCreated, lifecycle.whenStarted, and lifecycle.whenResumed. Any coroutine run inside these blocks is suspended if the Lifecycle isn't at least in the minimal desired state.

```
class MyFragment : Fragment() {
    init {
        lifecycleScope.launch { this: CoroutineScope
            whenCreated { }
            whenStarted { }
            whenResumed { }
        }
    }
}
```

```
class MyFragment: Fragment {
    init { // Notice that we can safely launch in the constructor of the Fragment.
        lifecycleScope.launch {
            whenStarted {
                // The block inside will run only when Lifecycle is at least STARTED
                // It will start executing when fragment is started and
                // can call other suspend methods.
                loadingView.visibility = View.VISIBLE
                val canAccess = withContext(Dispatchers.IO) {
                    checkUserAccess()
                }

                // When checkUserAccess returns, the next line is automatically
                // suspended if the Lifecycle is not *at least* STARTED.
                // We could safely run fragment transactions because we know the
                // code won't run unless the lifecycle is at least STARTED.
                loadingView.visibility = View.GONE
                if (canAccess == false) {
                    findNavController().popBackStack()
                } else {
                    showContent()
                }
            }

            // This line runs only after the whenStarted block above has completed
        }
    }
}
```

And our revised code will look like this (check it, kill your activity and see the exception is thrown):

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
    private val TAG = MainActivity::class.java.name

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // launch in main/ui thread
        // LifecycleOwner.LifecycleScope.launch { } - No need for LifecycleOwner cause it is this
        LifecycleScope.launch { this: CoroutineScope
            try {
                delay(Long.MAX_VALUE)
            } catch (e: Exception) {
                Log.d(TAG, msg: "coroutine cancelled")
                // e will be a JobCancellationException if the activity is destroyed
            }
        }

        LifecycleScope.launch { this: CoroutineScope
            fetchAndShowUser()
            delay( timeMillis: 700)
        }

        val job = LifecycleScope.launch (start = CoroutineStart.LAZY){ this: CoroutineScope
            println("Task preformed")
        }

        job.start()
    }
}
```

```
private suspend fun fetchAndShowUser() {
    val user1 = fetchUserOneAsync()
    Log.d(TAG, msg: "user one fetched")
    val user2 = fetchUserTwoAsync()
    Log.d(TAG, msg: "user two fetched")
    showUser(user1.await(),user2.await())
}

private fun showUser(user1:User, user2:User) {
    binding.textViewOne.text = user1.toString()
    binding.textViewTwo.text = user2.toString()
}

private fun fetchUserOneAsync() = LifecycleScope.async(Dispatchers.IO) { this: CoroutineScope
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe", email: "moshe@moshe.com") ^async
}

private fun fetchUserTwoAsync() = LifecycleScope.async(Dispatchers.IO) { this: CoroutineScope
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Misha", email: "misha@misha.com") ^async
}

}

data class User(val name:String, val email:String)
```

viewModelScope

Often if your ViewModel is destroyed, there's a bunch of "work" associated with the ViewModel that should be stopped as well. For example, let's say you're preparing a bitmap to show on-screen. That's an example of work you should do without blocking the main thread *and* work that should be stopped if you permanently navigate away from or close the screen. For work like this, you should use [viewModelScope](#).

viewModelScope is a Kotlin extension property on the ViewModel class. It is a CoroutineScope that is cancelled once the ViewModel is destroyed (when [onCleared\(\)](#) is called). Thus when you're using a ViewModel, you can start all of your coroutines using this scope.

For ViewModelScope, use androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1

Here is an example:

```
class MyViewModel : ViewModel() {  
  
    /**  
     * Heavy operation that cannot be done in the Main Thread  
     */  
    fun launchDataLoad() {  
        viewModelScope.launch {  
            sortList()  
            // Modify UI  
        }  
    }  
  
    suspend fun sortList() = withContext(Dispatchers.Default) {  
        // Heavy work  
    }  
}
```

Just think that the above code save all the code below:

```
class MyViewModel : ViewModel() {

    /**
     * This is the job for all coroutines started by this ViewModel.
     * Cancelling this job will cancel all coroutines started by this ViewModel.
     */
    private val viewModelJob = SupervisorJob()

    /**
     * This is the main scope for all coroutines launched by MainViewModel.
     * Since we pass viewModelJob, you can cancel all coroutines
     * launched by uiScope by calling viewModelJob.cancel()
     */
    private val uiScope = CoroutineScope(Dispatchers.Main + viewModelJob)

    /**
     * Cancel all coroutines when the ViewModel is cleared
     */
    override fun onCleared() {
        super.onCleared()
        viewModelJob.cancel()
    }

    /**
     * Heavy operation that cannot be done in the Main Thread
     */
    fun launchDataLoad() {
        uiScope.launch {
            sortList() // happens on the background
            // Modify UI
        }
    }

    // Move the execution off the main thread using withContext(Dispatchers.Default)
    suspend fun sortList() = withContext(Dispatchers.Default) {
        // Heavy work
    }
}
```

More on Coroutines And View Model

<https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471>

(Also available testing coroutines with mockito)

<https://medium.com/androiddevelopers/viewmodels-with-saved-state-jetpack-navigation-data-binding-and-coroutines-df476b78144e>

LiveData Scope special use cases of coroutines and Live Data

When using **LiveData**, you might need to calculate values asynchronously. For example, you might want to retrieve a user's preferences and serve them to your UI. In these cases, you can use the `liveData` builder function to call a suspend function, serving the result as a LiveData object.


```
val user: LiveData<User> = LiveData {
    val data = database.loadUser() // loadUser is a suspend function.
    emit(data)
}
```

The code block starts executing when LiveData becomes active and is automatically canceled after a configurable timeout when the LiveData becomes inactive. If it is canceled before completion, it is restarted if the LiveData becomes active again. If it completed successfully in a previous run, it doesn't restart.

You can also emit multiple values from the block. **Each emit() call suspends the execution of the block until the LiveData value is set on the main thread.**

```
val user: LiveData<Result> = LiveData {
    emit(Result.loading())
    try {
        emit(Result.success(fetchUser()))
    } catch (ioException: Exception) {
        emit(Result.error(ioException))
    }
}
```

You can emit multiple values from a LiveData by calling the emitSource() function whenever you want to emit a new value. Note that each call to emit() or emitSource() removes the previously-added source.

This means that you can use emit whenever you want to set a value once, but if you want to connect your live data to another live data value you use emit source.

```
liveData{
    emit(db.getData())
    val latest = webService.getLatestData()
    db.insert(latest)
    emit(db.getData())
}
```

But with emitSource it looks like this:

```
liveData{
    emitSource(db.getData())
    val latest = webService.getLatestData()
    db.insert(latest)
}
```

Don't need to call emit again since the liveData already have a source.

suspendCoroutine

Obtains the current continuation instance inside suspend functions and suspends the currently running coroutine.

This is usually done to prevent nesting of callbacks and use a single suspended function instead

```
suspend fun <T> awaitTask(task: Task<T>): T = suspendCoroutine { continuation ->
    task.addOnCompleteListener { task ->
        if (task.isSuccessful) {
            continuation.resume(task.result)
        } else {
            continuation.resumeWithException(task.exception!!)
        }
    }
}
```

When we use this code we can simply call awaitTask function and get the info result without any callback hassle from our side.

The block of code passed to suspendCoroutine { ... } should not block a thread that it is being invoked on, allowing the coroutine to be suspended. This way, the actual thread can be used for other tasks. This is a key feature that allows Kotlin coroutines to scale and to run multiple coroutines even on the single UI thread.

Launchers, Providers & Location

Download the Launchers and Providers PDF Guide:

https://drive.google.com/file/d/1QVw_1jvaa4IO0ldZ3KFM1okWoCGF1N5B/view?usp=sharing

Download the Launchers and Providers Startup Files:

https://drive.google.com/file/d/1OMSbboOlzCBvCT7VE_fQrnZQRQGouKPc/view?usp=sharing

Download The Launchers and Providers full App Created in the Guide:

<https://drive.google.com/file/d/1EvESUlnFAVyik2aWrUqNstBch-f8mPKU/view?usp=sharing>

Download the Location App Dependencies PDF:

<https://drive.google.com/file/d/1w6j5AcsCBJSVHiaBblqufLDq1EJsaTWG/view?usp=sharing>

Download the Location App created in the Videos:

https://drive.google.com/file/d/1WVfzHR2aWcv2_-hxMBgURJ-hC8Gj7ilq/view?usp=sharing

Download the Contacts App Starter files:

https://drive.google.com/file/d/1u3H9MYG_H9M_uDNI1wQd0hpuwNocjHdk/view?usp=sharing

Download the Contacts App Dependencies PDF:

<https://drive.google.com/file/d/1QijLRWRrQZ6UbbNp8GRkGL4lxGXzCeGG/view?usp=sharing>

Download the Contacts Full App created in the videos:

<https://drive.google.com/file/d/1moyGiOBSIVyz83LUjMjvLknnBKmJtuG7/view?usp=sharing>

The new Launchers API + Working with FileProvider

AndroidX introduced a new API for opening an Activity or requesting a permission and getting back a result from them.

First let's understand that the old way is now deprecated. In the old way we used to give each activity or permission request a different code cause they all returned to same overridden function (you can only override a function one time...) and we had to distinguish from where we have just returned from with the code checking - that caused a messy and C type coding, it violated the single responsibility cause the same function is responsible for a lot of different actions, its confusing and additionally, if another similar functionality screen appears in the app, we won't be able to reuse our code and will have to duplicate it.

For that reason JetPack introduced the new Activity for result API

The activity for result new Api based on these three steps:



Step 1

Contract is a class that implements the `ActivityResultContract<I,O>` interface. Where I define the type of input data necessary to start the Activity, and O defines the callback result type.

The great thing is that unlike the old api here we have pre-made contracts that fill the most common tasks like taking a picture and getting the thumbnail or the full photo back, picking an image from the gallery, picking a contact and much more. Here you can find a list of the contracts :

<https://developer.android.com/reference/androidx/activity/result/contract/ActivityResultContracts>

we can also use the pre-made general contract who looks similar to the old api. Later we see how to create a contract yourself and how to work with the general contract but for now we will use the pre-made contracts.

Step 2

The next step is to register the contract in the activity or fragment by calling `registerForActivityResult()`. You need to pass `ActivityResultContract` and `ActivityResultCallback` as parameters. The callback will be invoked when the result is received. It is important to register the **contract before the activity or fragment is created.**

This is the official reason for the new api. Google wanted to decouple the launching from the registering because sometimes due to low memory conditions our process or activity can get destroyed, and the result callback needs to be available when your process and activity are recreated, for that reason the callback must be unconditionally registered every time your activity is created.



Step 3

To start the Activity for result, we only need to call `launch()` on the `ActivityResultLauncher` object that we have obtained in the previous step.

Build a simple usage app

Open a new Android studio project and copy the xml files from the starters folder.

Bind your views and return the root and create also the `TestActivity` and bind it's views.

Add the following dependencies to your app grade file:

```
implementation 'androidx.core:core-ktx:1.6.0'
```

```
implementation 'com.github.bumptech.glide:glide:4.12.0'
annotationProcessor 'com.github.bumptech.glide:compiler:4.12.0'
```

Lets look at those three stages with the most common contracts:

Step 2 - Register - (No need for Step 1 because the contact already exists) When Activity created

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding : ActivityMainBinding

    val cameraResultLauncher: ActivityResultLauncher<Void> =
        registerForActivityResult(TakePicturePreview()) { it: Bitmap!
            binding.resultIv.setImageBitmap(it)
        }

    val cameraFullSizeResultLauncher: ActivityResultLauncher<Uri> =
        registerForActivityResult(ActivityResultContracts.TakePicture()) { it: Boolean!

        }

    val pickContentResultLauncher: ActivityResultLauncher<Array<String>> =
        registerForActivityResult(ActivityResultContracts.OpenDocument()) { it: Uri!
            binding.resultIv.setImageURI(it)
            //if you want to save it for later use
            contentResolver.takePersistableUriPermission(it, modeFlags: Intent.FLAG_GRANT_READ_URI_PERMISSION
                or Intent.FLAG_GRANT_WRITE_URI_PERMISSION)
            getSharedPreferences( name: "details", MODE_PRIVATE).edit (commit = true){ this: SharedPreferences.Editor
                putString("pic_uri",it.toString())
            }
        }
}
```

Step 3 - Launching

When pressing the button for example

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    binding.picBtn.setOnClickListener { it: View!
        cameraResultLauncher.launch( input: null)
    }

    binding.galleryBtn.setOnClickListener { it: View!
        pickContentResultLauncher.launch(arrayOf("image/*"))
    }
}
```

There is also the General contract - we can use it in one time cases where there isn't a pre-defined contract for our action and we don't want to define our own contract - lets demonstrate with speech recognition:

```
val recognizeSpeechResultLauncher : ActivityResultLauncher<Intent> =
    registerForActivityResult(ActivityResultContracts.StartActivityForResult()) { it: ActivityResult!
        if(it.resultCode == RESULT_OK)
            binding.textView.text = it.data?.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS).toString()
    }
}
```

```
binding.speechBtn.setOnClickListener { it: View!
    val intent = Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH).apply { this: Intent
        putExtra(RecognizerIntent.EXTRA_MAX_RESULTS, value: 5)
        putExtra(RecognizerIntent.EXTRA_PROMPT, value: "Speak slowly please")
        putExtra(RecognizerIntent.EXTRA_LANGUAGE, value: "iw")
    }
    recognizeSpeechResultLauncher.launch(intent)
}
```

Create your own contract

When creating a contract, we should implement two of its methods:

- *createIntent()* — accepts input data and creates an intent, which will be later launched by calling *launch()*
- *parseResult()* — is responsible for returning the result, handling *resultCode*, and parsing the data.

Another method, *getSynchronousResult()*, can be overridden if necessary. It allows you to return the result immediately, without starting the Activity, for example, if the received input data is invalid. If this behavior is not required, the method returns null by default.

For example lets look at a test activity that receives the user name and return his grade:

The Activity:

```
class TestActivity : AppCompatActivity() {

    private lateinit var binding : ActivityTestBinding
    private var q1 : Boolean = false
    private var q2 : Boolean = false

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityTestBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.nameOutput.text = intent.getStringExtra( name: "name")
        binding.rgq1.setOnCheckedChangeListener { radioGroup, i ->
            q1 = when(i) { R.id.rb3q1 -> true else -> false }
        }
        binding.rgq2.setOnCheckedChangeListener { radioGroup, i ->
            q2 = when(i) { R.id.rb2q2 -> true else -> false }
        }
        binding.finishBtn.setOnClickListener { it: View!

            var grade = 0
            if(q1) grade+=50
            if(q2) grade+=50
            val intent = Intent().apply { putExtra( name: "grade", grade) }
            setResult(RESULT_OK,intent)
        }
    }
}
```

You need to create the following contract that receives a String the tester name and returns Int which is his grade

```
class MyTestActivityContract : ActivityResultContract<String,Int?>() {

    override fun createIntent(context: Context, input: String?): Intent {
        return Intent(context,TestActivity::class.java)
            .putExtra( name: "name",input)
    }

    override fun parseResult(resultCode: Int, intent: Intent?): Int? = when {
        resultCode != Activity.RESULT_OK -> null
        else -> intent?.getIntExtra( name: "grade", defaultValue: 0)
    }

    override fun getSynchrounousResult(context: Context, input: String?): SynchronousResult<Int?>? {
        return if (input.isNullOrEmpty()) SynchronousResult( value: 0) else null
    }
}
```

And now we can register a contract like this:


```
val takeTestResultLauncher : ActivityResultLauncher<String> =
    registerForActivityResult(MyTestActivityContract()) { it: Int?
        when(it) {
            null -> binding.textView.text = "Test not finished"
            else -> binding.textView.text = "Your grade is $it"
        }
    }
```

And execute it like this:

```
binding.takeTestBtn.setOnClickListener { it: View!
    takeTestResultLauncher.launch(binding.testNameEt.text.toString())
}
```

Getting the full size photo using FileProvider

To get a full size photo we need to supply a URI for the system to save the photo taken. But there is a problem.

When writing this:

```
binding.fullSizePicBtn.setOnClickListener { it: View!
    file = File(getExternalFilesDir(Environment.DIRECTORY_PICTURES), child: "temp.jpg")
    cameraFullSizeResultLauncher.launch(Uri.fromFile(file))
}
```

Here we gave out path to our app folder in a he external storage. Writing to this path don't require permission, all files in this folder can be shared with others and they will be removed when the user uninstall our app.

And add the following code to you result callback.

```
val cameraFullSizeResultLauncher: ActivityResultLauncher<Uri> =
    registerForActivityResult(ActivityResultContracts.TakePicture()) { it: Boolean!
        if(it){
            Glide.with(activity: this).load(file).into(binding.resultIv)
            //or without Glide
            // binding.resultIv.setImageBitmap(BitmapFactory.decodeFile(file.absolutePath))
        }
    }
```

When executing this code we get the following error:

```
android.os.FileUriExposedException: file:///storage/emulated/0/Android/data/il.co.syntax.activityresultlauncherkotlin/files/Pictures/temp.jpg
exposed beyond app through ClipData.Item.getUri()
```

This as Android exception that was added in API 24 and it meaning that application exposes a file:// uri to another app.

We are here by exposing a path of our file system to another app that may have

not asked for the READ_EXTERNAL_STORAGE permission and it can potentially read the data from there even long after her task of saving the image had finished.

One bad solution to "solve" this problem as to change the policy. Like any other Android exception we can simply change the strict policy and all is solved:

```
val builder = VmPolicy.Builder()  
StrictMode.setVmPolicy(builder.build())
```

Really? Do you seriously think that this is the answer to our problem. The risk stays, we didn't solve it just bypassed it.

So remove these lines and understand the real solution:

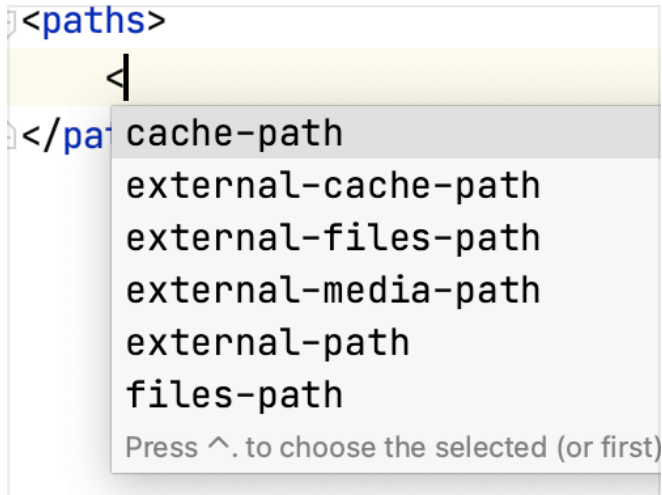
The real solution is to provide a temporary uri through FileProvider which we must add to our manifest like any other component. We must allow this FileProvider to create URI's and even tell him where he can do this(in which directories) only then we can request it to create a temporary uri starting with "content://" this is uri is only valid until the receiving app is destroyed. After that this uri is no longer valid and points to nothing!

Content providers are one of the primary building blocks of Android applications, providing content to applications. They encapsulate data and provide it to applications through the single ContentResolver interface. A content provider is only required if you need to share data between multiple applications. For example, the contacts data is used by multiple applications and must be stored in a content provider. If you don't need to share data amongst multiple applications you can use a database directly via SQLiteDatabase.

Here are the full stages:

In the resource folder create the xml directory if not exists and create an xml file with the <paths> root element

This file will contain the paths that the provider can create temporary uri into



The `<paths>` element must contain one or more of the following child elements:

```
<files-path name="name" path="path" />
```

Represents files in the `files/` subdirectory of your app's internal storage area. This subdirectory is the same as the value returned by `Context.getFilesDir()`.

```
<cache-path name="name" path="path" />
```

Represents files in the cache subdirectory of your app's internal storage area. The root path of this subdirectory is the same as the value returned by `getCacheDir()`.

```
<external-path name="name" path="path" />
```

Represents files in the root of the external storage area. The root path of this subdirectory is the same as the value returned by `Environment.getExternalStorageDirectory()`.

```
<external-files-path name="name" path="path" />
```

Represents files in the root of your app's external storage area. The root path of this subdirectory is the same as the value returned by `Context#getExternalFilesDir(String)` `Context.getExternalFilesDir(null)`.

```
<external-cache-path name="name" path="path" />
```

Represents files in the root of your app's external cache area. The root path of this subdirectory is the same as the value returned by `Context.getExternalCacheDir()`.

```
<external-media-path name="name" path="path" />
```

Represents files in the root of your app's external media area. The root path of this subdirectory is the same as the value returned by the first result of `Context.getExternalMediaDirs()`.

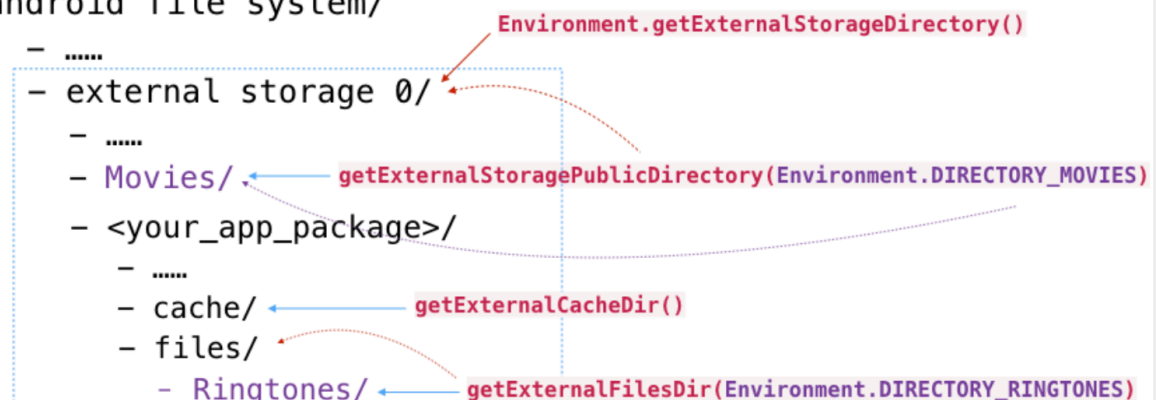
```
name="name"
```

A URI path segment. To enforce security, this value hides the name of the subdirectory you're sharing. The subdirectory name for this value is contained in the `path` attribute.

```
path="path"
```

The subdirectory you're sharing. While the `name` attribute is a URI path segment, the `path` value is an actual subdirectory name. Notice that the value refers to a **subdirectory**, not an individual file or files. You can't share a single file by its file name, nor can you specify a subset of files using wildcards.

android file system/



In our case we will save the file to the external storage of the application - it will be deleted when we uninstall our app but we can use it to store our own files and share them with others. Writing or reading to or from this folder doesn't require any special permissions since Android 4.4

In particular we will add the picture to the Pictures subdirectory and we share it as images (to hide the actual path)

```
<paths>
  <external-files-path name="my_images" path="Pictures"/>
</paths>
```

Now we need to Add a FileProvider <provider> tag in AndroidManifest.xml under <application> tag. Specify a unique authority for the android:authorities attribute to avoid conflicts(use your package). Also allow him to grant URI permission and specify the location of the xml file coating the file paths you just created.

```
<provider
  android:name="androidx.core.content.FileProvider"
  android:authorities="${applicationId}.provider"
  android:exported="false"
  android:grantUriPermissions="true">
  <meta-data
    android:name="android.support.FILE_PROVIDER_PATHS"
    android:resource="@xml/provider_paths" />
</provider>
```

android:authorities A list of one or more URI authorities that identify data offered by the content provider.

The Android system stores references to content providers according to an **authority** string, part of the provider's **content URI that will be created by him**.

For example lets look of the generated uri:

content://il.co.syntax.activityresultlauncherkotlin.provider/my_images/temp.jpg

The content: **scheme** identifies the URI as a content URI pointing to an Android content provider. The authority il.co.syntax.activityresultlauncherkotlin.provider identifies the provider itself; The substring /my_images/temp.jpg is a **path hidden by the name attribute**), which the content provider can use to identify subsets of the provider data.

We have just Declared a content provider component. A content provider is a subclass of **ContentProvider** that supplies structured access to data managed by the application. All content providers in your application must be defined in a <provider> element in the manifest file; otherwise, the system is unaware of them and won't run them.

You only declare content providers that are part of your application. Content providers in other applications that you use in your application should not be declared.

The last thing is to ask the File Provider to create the temporary URI:

```
binding.fullSizePicBtn.setOnClickListener { it: View!
    file = File(getExternalFilesDir(Environment.DIRECTORY_PICTURES), child: "temp.jpg")

    val photoURI = FileProvider.getUriForFile(
        context: this,
        authority: "$packageName.provider",
        file
    )

    cameraFullSizeResultLauncher.launch(photoURI)
}
```

That's it.

Runtime Permissions with Launchers

Requesting permission or permission is just another pre-made contract we have!

Lets say that in the previous example we wanted to save our image in the external storage that is shared for all apps, for example we want the picture to be added to the gallery and we don't want it to be deleted when the user removes the app. This place:

```
<external-path name="my_images" path="Pictures"/>
```

Now we would have needed WRITE_EXTERNAL_STORAGE permission (it grant us the also READ permission implicitly), let's look at the new jetpack launcher tool that helps us also here (The old permission way was exactly the same as the old activity for result).

First initialize your launchers with the pre-made contracts:

```
private val permission : ActivityResultLauncher<String>
= registerForActivityResult(ActivityResultContracts.RequestPermission()) { granted ->
    when {
        granted-> {
            //We have the permission! Here you should start the full size photo launcher
            //what was previously written in the button click
        }
        Build.VERSION.SDK_INT>=23 &&
            !shouldShowRequestPermissionRationale(Manifest.permission.WRITE_EXTERNAL_STORAGE)-> {
            //The permission was denied and the user clicked the don't ask again checkbox
        }
        else -> {
            //the permission was denied by the user
        }
    }
}
```

Note: we have also the ActivityResultContracts.RequestMultiplePermissions() that returns not ActivityResultLauncher<String>

But a `ActivityResultLauncher<Array<out String>>` and the lambda expression is not boolean but a map of string and boolean. And of course when we launch it we pass an array of string and not a single one.

Here is how we launch our permission request:

```
binding.fullSizePicBtn.setOnClickListener { it: View!  
  
    if (Build.VERSION.SDK_INT >= 23 && shouldShowRequestPermissionRationale(Manifest.permission.WRITE_EXTERNAL_STORAGE)  
        // access to the external storage is prohibited, explain why the permission is required  
    ) else {  
        permission.launch(Manifest.permission.CAMERA)  
    }  
}
```

And put your full size camera launcher in the granted section

<https://developer.android.com/training/basics/intents/result>

<https://medium.com/e-legion/the-right-way-to-get-a-result-part-i-activity-result-api-6efbcaa5600d>

Receiving an activity result in a separate class

While the `ComponentActivity` and `Fragment` classes implement the `ActivityResultCaller` interface to allow you to use the `registerForActivityResult()` APIs, you can also receive the activity result in a separate class that does not implement `ActivityResultCaller` by using `ActivityResultRegistry` directly. For example, you might want to implement a `LifecycleObserver` that handles registering a contract along with launching the launcher:

```
class MyLifecycleObserver(private val registry : ActivityResultRegistry)
    : DefaultLifecycleObserver {
    lateinit var getContent : ActivityResultLauncher<String>

    override fun onCreate(owner: LifecycleOwner) {
        getContent = registry.register("key", owner, GetContent()) { uri ->
            // Handle the returned Uri
        }
    }

    fun selectImage() {
        getContent.launch("image/*")
    }
}

class MyFragment : Fragment() {
    lateinit var observer : MyLifecycleObserver

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...

        observer = MyLifecycleObserver(requireActivity().activityResultRegistry)
        lifecycle.addObserver(observer)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        val selectButton = view.findViewById<Button>(R.id.select_button)

        selectButton.setOnClickListener {
            // Open the activity to select an image
            observer.selectImage()
        }
    }
}
```

When using the ActivityResultRegistry APIs, it's strongly recommended to use the APIs that take a LifecycleOwner, as the LifecycleOwner automatically removes your registered launcher when the Lifecycle is destroyed. However, in cases where a LifecycleOwner is not available, each ActivityResultLauncher class allows you to manually call `unregister()` as an alternative.

Firestore MVVM

Download the App Starter Files:

<https://drive.google.com/file/d/1tzmmt0sv0LwVH1-SGYq2YaPRpJhVh5uT/view?usp=sharing>

Download the Starter project(optional):

<https://drive.google.com/file/d/1QVqTadZQ9wLvyUHH-u8iNle3nXy3tQ3u/view?usp=sharing>

Download the Full Firestore App Created in the videos:

https://drive.google.com/file/d/1DvSgazhaCvSHpxF__v-3YZanidFn2sRp/view?usp=sharing

Dependency Injection with Hilt

Download the full App Created in this Guide:

<https://drive.google.com/file/d/1C2mejEMsalzfdJZ8BtF3WTCUv7SCKFLj/view?usp=sharing>

A dependency is an object that another object requires. In other words, the latter object depends on the former for it to function. For example, a Car class might need a reference to an Engine class.

There are three ways for a class to get an object it needs:

1. The class constructs the dependency it needs. In the example above, Car would create and initialize its own instance of Engine.
2. Grab it from somewhere else. Some Android APIs, such as Context getters and getSystemService(), work this way.
3. Have it supplied as a parameter. The app can provide these dependencies when the class is constructed or pass them in to the functions that need each dependency. In the example above, the Car constructor would receive Engine as a parameter - **this dependency injection!** With this approach you take the dependencies of a class and provide them rather than having the class instance obtain them herself.

Dependency Injection is whereby dependencies are provided to a class instead of the class having to create them itself. Hilt is a standardized way of enforcing dependency injection in an Android application.

In the first two options Car and Engine are tightly coupled - an instance of Car uses one type of Engine, **and no subclasses or alternative implementations can easily be used**. If the Car were to construct its own Engine, you would have to create two types of Car instead of just reusing the same Car for engines of type Gas and Electric. It also makes the tests much harder because **Car must have a real instance of engine thus preventing us from the ability to mock it**.

Without dependency injection:

```
class Car {
    private val engine = Engine()

    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val car = Car()
    car.start()
}
```

With dependency injection

```
class Car(private val engine: Engine) {
    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val engine = Engine()
    val car = Car(engine)
    car.start()
}
```

Write these classes in a new android studio project.

Lets look at the Reusability of Car:

You can pass in different implementations of Engine to Car. For example, you might define a new subclass of Engine called ElectricEngine that you want Car to use. If you use DI, all you need to do is pass in an instance of the updated ElectricEngine subclass, and Car still works without any further changes.

```
class Car(val engine: Engine) {
    fun start(){
        engine.start();
    }
}

open class Engine {
    open fun start() {
        println("engine started")
    }
}

class ElectricEngine:Engine() {
    override fun start() {
        println("electric engine started")
    }
}
```

```
val e = Engine()
val e1 = ElectricEngine()
val c = Car(e)
val ec = Car(e1)
c.start()
ec.start()
```

There are two major ways to do dependency injection in Android:

- **Constructor Injection.** This is the way described above. You pass the dependencies of a class to its constructor.

```
class Car(val engine: Engine) {  
  
    // val engine = Engine()  
  
    fun drive() {  
        Log.d(tag: "CarHilt", msg: "We drive the car")  
    }  
}  
  
class Engine() {  
  
    fun checkOil() {  
        Log.d(tag: "EngineHilt", msg: "Oil checked")  
    }  
}
```

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        val engine = Engine()  
        val car = Car(engine)  
        car.engine.checkOil()  
        car.drive()  
    }  
}
```

- **Field Injection (or Setter Injection).** Certain Android framework classes such as activities and fragments are instantiated by the system, so constructor injection is not possible. With field injection, dependencies are instantiated after the class is created. The code would look like this:

```
class Car {
    lateinit var engine: Engine

    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val car = Car()
    car.engine = Engine()
    car.start()
}
```

In all the examples above we did the dependency injection manually

Automated dependency injection

In the previous example, you created, provided, and managed the dependencies of the different classes yourself, without relying on a library. This is called *dependency injection by hand*, or **manual dependency injection**. In the Car example, there was only one dependency, but more dependencies and classes can make manual injection of dependencies more tedious. Also When you're not able to construct dependencies before passing them in — for example when using lazy initializations — you need to write and maintain a custom container (or graph of dependencies) that manages the lifetimes of your dependencies in memory.

Look at this example to see fully manual dependency injection

<https://developer.android.com/training/dependency-injection/manual>

There are libraries that solve this problem by automating the process of creating and providing dependencies. They fit into two categories:

- Reflection-based solutions that connect dependencies at runtime. Example of this solution is using the **Guice library**.
- Static solutions that generate the code to connect dependencies at compile time. Example is using Dagger2 library which is now managed by Google and a jetpack composed library called **Hilt**.

Dagger is a popular dependency injection library for Java, Kotlin, and Android that is maintained by Google. Dagger facilitates using DI in your app by creating and managing the graph of dependencies for you. It provides fully static and

compile-time dependencies

Use Hilt in your Android app

Hilt is Jetpack's recommended library for dependency injection in Android. Hilt defines a standard way to do DI in your application by **providing containers for every Android class in your project and managing their lifecycles automatically for you.**

Hilt is built on top of the popular DI library **Dagger** to benefit from the compile time correctness, runtime performance, scalability, and Android Studio support that Dagger provides.

Using Hilt

First, add the hilt-android-gradle-plugin plugin to your **project's root** build.gradle file:

```
dependencies {  
    ...  
    classpath 'com.google.dagger:hilt-android-gradle-plugin:2.38.1'  
}
```

Then, apply the Gradle plugin and add these dependencies in your app/ build.gradle file:

```
id 'kotlin-kapt'  
id 'dagger.hilt.android.plugin'
```

```
dependencies {  
    implementation "com.google.dagger:hilt-android:2.38.1"  
    kapt "com.google.dagger:hilt-compiler:2.38.1"  
}
```

Make sure Java 8 is enabled (Hilt uses Java 8) - in you app Gradle file

```
compileOptions {  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}
```

First step

Extend the Application class and annotate it with **@HiltAndroidApp** this triggers Hilt's code generation, including a base class for your application that serves as the application-level dependency container.

```
@HiltAndroidApp
class MyApplication : Application() {
}
```

Don't forget to add the application name to the manifest file

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="HiltExample"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportRtl="true"
    android:theme="@style/Theme.HiltExample"
    android:name=".MyApplication">
```

This generated Hilt component is attached to the Application object's lifecycle and provides dependencies to it. Additionally, it is the parent component of the app, which means that other components can access the dependencies that it provides.

Once Hilt is set up in your Application class and an application-level component is available, Hilt can provide dependencies to other Android classes that have the `@AndroidEntryPoint` annotation:

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
```

Hilt currently supports the following Android classes:

- Application (by using `@HiltAndroidApp`)
- ViewModel (by using `@HiltViewModel`)
- Activity
- Fragment
- View
- Service
- BroadcastReceiver

If you annotate an Android class with `@AndroidEntryPoint`, then you also must annotate Android classes that depend on it. For example, if you annotate a fragment, then you must also annotate any activities where you use that fragment. Classes that Hilt injects can have other base classes that also use injection. Those classes don't need the `@AndroidEntryPoint` annotation if they're abstract.

@AndroidEntryPoint generates an individual Hilt component for each Android class in your project. It turns them into dependency containers.

Define Hilt bindings

To perform field injection, Hilt needs to know how to provide instances of the necessary dependencies from the corresponding component. An **Hilt binding** contains the information necessary to provide instances of a type as a dependency.

One way to provide binding information to Hilt is *constructor injection*. Use the **@Inject** annotation on the constructor of a class to tell Hilt how to provide instances of that class.

Our full code will look like this now:

```
class Car @Inject constructor(val engine: Engine) {  
  
    // val engine = Engine()  
    fun drive() {  
        Log.d(tag: "CarHilt", msg: "We drive the car")  
    }  
}  
  
class Engine @Inject constructor() {  
  
    fun checkOil() {  
        Log.d(tag: "EngineHilt", msg: "Oil checked")  
    }  
}
```

Here **@Inject** gives Hilt access to the necessary constructors meaning it can generate instances of both Car and Engine. If Engine is a parameter of the Injected constructor then Hilt must also know how to create instances of it (the Engine class).

Instances that Hilt knows how to create go by the name bindings. So Car and Engine are bindings.

And the Activity which is holding the Injectable Fields will eventually look like this:

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {

    @Inject lateinit var car : Car

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        car.engine.checkOil()
        car.drive()
    }
}
```

To obtain dependencies from a component, use the **@Inject** annotation to perform field injection.

Here the **@Inject** annotation goes by a different meaning. Here it means the car field is injectable field. Injectable means that Hilt can supply the instantiated dependencies to it.

Please note that Fields injected by Hilt cannot be private. Attempting to inject a private field with Hilt results in a compilation error.

Hilt modules

Sometimes a type cannot be constructor-injected. This can happen for multiple reasons. For example, you cannot constructor-inject an interface. You also cannot constructor-inject a type that you do not own, such as a class from an external library. In these cases, you can provide Hilt with binding information by using *Hilt modules*.

A Hilt module is a class that is annotated with **@Module** it informs Hilt how to provide instances of certain types. you must annotate Hilt modules with **@InstallIn** to tell Hilt which Android class each module will be used or installed in. This determine the dependency lifetime scope.

If you want the dependency to exist in all of your app activities use **@InstallIn(ActivityComponent::class)**. Later we will see all the available scopes.

Hilt can't generate a constructor for an interface. Instead, provide Hilt with the binding information by creating an abstract function annotated with **@Binds** inside a Hilt module.

The @Binds annotation tells Hilt which implementation to use when it needs to provide an instance of an interface.

The annotated function provides the following information to Hilt:

- The function return type tells Hilt what interface the function provides instances of.
- The function parameter tells Hilt which implementation to provide.

Therefore our code will look like this:

```
class Engine @Inject constructor() : Recyclable{

    fun checkOil() {
        Log.d( tag: "EngineHilt", msg: "Oil checked")
    }

    override fun recycle() {
        Log.d( tag: "EngineHilt", msg: "Engine recycled")
    }
}

interface Recyclable {
    fun recycle()
}

@Module
@InstallIn(ActivityComponent::class)
abstract class RecyclableModule {

    @Binds
    abstract fun EngineImpl(engine: Engine) : Recyclable
}
```

In our example when someone requires a Recyclable then we will return an Engine

And in the Main Activity we change the code like that:

```
@Inject lateinit var engine : Recyclable

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    engine.recycle()
}
}
```

Now what happens if we want different implementation for car and for engine. If we try to add the following function to our Module and run your code.

```
@Binds
abstract fun CarImpl(car: Car) : Recyclable
```

And of course make car also implement the Recyclable interface

```
class Car @Inject constructor(val engine: Engine) : Recyclable {
    override fun recycle() {
        println("car recycled")
    }
}
```

We will get the following error when we try to execute our code:

```
[Dagger/DuplicateBindings] il.co.syntax.hiltexample.Recyclable is bound multiple times:
```

Hilt doesn't know which implementation to use and We need to differentiate them somehow.

This is why we have the @Qualifier Annotation

Create the following Annotation Qualifiers next to your Module they will use you to differentiate the implementations

```
@Qualifier  
annotation class EngineQualifier  
  
@Qualifier  
annotation class CarQualifier
```

And add the Qualifier next to the implementations like this:

```
@Module  
@InstallIn(ActivityComponent::class)  
abstract class RecyclableModule {  
  
    @EngineQualifier  
    @Binds  
    abstract fun engineImpl(engine: Engine) : Recyclable  
  
    @CarQualifier  
    @Binds  
    abstract fun carImpl(car: Car) : Recyclable  
}
```

Also add them next to the reference definition in you MainActivity file

```
@EngineQualifier
@Inject lateinit var engine : Recyclable

@CarQualifier
@Inject lateinit var car : Recyclable

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    engine.recycle()
    car.recycle()
}
```

Interfaces are not the only case where you cannot constructor-inject a type. Constructor injection is also not possible if you don't own the class because it comes from an external library (classes like [Retrofit](#) or [Room databases](#)), or if instances must be created with the [builder pattern](#).

We can tell Hilt how to provide instances of a type by creating a function inside a Hilt module and annotating that function with **@Provides**.

Lets take for example a simple library called Gson to covert string to json and vice versa

Add the following dependency to your project

```
implementation 'com.google.code.gson:gson:2.8.6'
```

And create the following Gson module:

```
@Module
@InstallIn(ActivityComponent::class)
object GsonModule {
    @Provides
    fun provideGson(): Gson {
        return Gson()
    }
}
```

Through @Provides, the annotated function gives Hilt the following information:

- The return type tells Hilt what type the function provides instances of.
- The parameters tell Hilt the dependencies required to provide the type. In our case, there are none.
- The function body tells Hilt how to provide an instance of the corresponding type. Hilt executes the function body every time it needs to provide an instance of that type.

In the MainActivity add the following code

```
@Inject lateinit var gson: Gson

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    engine.recycle()
    car.recycle()

    Log.d(tag: "GsonHilt", gson.toString())
}
```

We have successfully injected 3rd library dependency!

A bit more about @Binds and @Provides :

Why is @Binds different from @Provides?

@Provides, the most common construct for configuring a binding, serves three functions:

1. Declare which type (possibly qualified) is being provided – this is the return type
2. Declare dependencies – these are the method parameters
3. Provide an implementation for exactly *how* the instance is provided – this is the method body

While the first two functions are unique and critical to every @Provides method, the third can often be tedious and repetitive. So, whenever there is a @Provides whose implementation is simple and common enough to be inferred by Dagger, it makes sense to just declare that as a method without a body (an abstract method) and have Dagger apply the behavior.

But, if we were to just say that abstract @Provides methods should be treated as we do for @Binds methods, the specification of @Provides would basically be two specifications with a bunch of conditional logic. For example, a @Provides method can have any number of parameters of any type, but a @Binds method can only have a single parameter whose type is assignable to the return type. Separating those specifications makes it easier to reason about correctness because the annotation determines the constraints.

Why can't @Binds and instance @Provides methods go in the same module?

Because @Binds methods are *just* a method *declaration*, they are expressed as abstract methods – no implementation is ever created and nothing is ever invoked. On the other hand, a @Provides method *does* have an implementation and *will* be invoked.

Since @Binds methods are never implemented, no concrete class is ever created that implements those methods. However, instance @Provides methods *require* a concrete class in order to construct an instance on which the method can be invoked.

What do I do instead?

The easiest change is to make the provides method `static`. In addition to being compatible with @Binds, they often perform better than instance provides methods.

If the method *must* be an instance method (e.g. returns a value from a field), the easiest fix is to separate your @Provides methods and @Binds methods into two separate modules and include one from the other. A simple example that provides an

As you can see @Binds functions are abstract while @Provides functions have a body. With @Binds the implementation is obvious. @Binds method can only have a single parameter whose type is assignable to the return type. @Provides method can have any number of parameters of any type.

More on @InstallIn

Now let's look at the scope again, try writing the same lines in you Application class. You will get an error!!

```
@Inject
lateinit var gson: Gson

@HiltAndroidApp
class MyApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        Log.d( tag: "GsonHilt",gson.toString())
    }
}
```

Do you remember the discussion on components? If you look at the GsonModule component, it is installed in the ActivityComponent.class. Therefore, it is only available during the lifetime of an activity rather than that of the entire application.

For each Android class in which you can perform field injection, there's an associated Hilt component that you can refer to in the @InstallIn annotation. Each Hilt component is responsible for injecting its bindings into the corresponding Android class.

The previous examples demonstrated the use of ActivityComponent in Hilt modules.

Hilt provides the following components:

Hilt component	Injector for
SingletonComponent	Application
ActivityRetainedComponent	N/A
ViewModelComponent	ViewModel
ActivityComponent	Activity
FragmentComponent	Fragment
ViewComponent	View
ViewWithFragmentComponent	View annotated with @WithFragmentBindings
ServiceComponent	Service

Hilt automatically creates and destroys instances of generated component classes following the lifecycle of the corresponding Android classes (for example, all the activity components will be destroyed by hilt in the activity onDestroy() method)

When it comes to classes such as Gson, Retrofit and Room database, we may need to make them available to the entire application.

To correct this error, change the ActivityComponent.class to SingletonComponent.class - The error is gone

```
@Module
@InstallIn(SingletonComponent::class)
object GsonModule {
```

But is the Gson object the same in MyApplication and MainActivity? No.

Scoping

Bindings in Hilt are naturally **unscoped**. This means that each time your app requests the binding (the dependency), **Hilt creates a new instance of the needed type.**

However, Hilt also allows a binding to be scoped to a particular component. Hilt only creates a scoped binding once per instance of the component that the binding is scoped to, and all requests for that binding share the same instance.

The table below lists scope annotations for each generated component:

Android class	Generated component	Scope
Application	SingletonComponent	@Singleton
Activity	ActivityRetainedComponent	@ActivityRetainedScoped
ViewModel	ViewModelComponent	@ViewModelScoped
Activity	ActivityComponent	@ActivityScoped
Fragment	FragmentComponent	@FragmentScoped
View	ViewComponent	@ViewScoped
View annotated with @WithFragmentBindings	ViewWithFragmentComponent	@ViewScoped
Service	ServiceComponent	@ServiceScoped

To ensure only one instance of Gson is available at a time, modify GsonModule and add @Singleton annotation used to ensure that the generated instance is the only one throughout the application's lifecycle.

```
@Module
@InstallIn(SingletonComponent::class)
object GsonModule {

    @Singleton
    @Provides
    fun provideGson(): Gson {
        return Gson()
    }
}
```

Because we scoped the GsonModule to the SingletonComponent using @Singleton Hilt provides the same instance of GsonModule throughout the life

of the entire application.

ActivityScoped ensures that the instance is the same throughout the activity. Same if we would have scoped any other adapter or module to the ActivityComponent using @ActivityScoped, then Hilt would have provided the same instance of that module throughout the life of the corresponding activity.

```
@ActivityScoped
class AnalyticsAdapter @Inject constructor(
    private val service: AnalyticsService
) { ... }
```

Please note that Scoping a binding to a component can be costly because the provided object stays in memory until that component is destroyed.

To summarize the interfaces and the 3rd library in oppose to interfaces or class we own but can't call their constructor - this it how to obtain a single instance of the AnalyticSercive:

```
// If AnalyticsService is an interface.
@Module
@InstallIn(SingletonComponent::class)
abstract class AnalyticsModule {

    @Singleton
    @Binds
    abstract fun bindAnalyticsService(
        analyticsServiceImpl: AnalyticsServiceImpl
    ): AnalyticsService
}

// If you don't own AnalyticsService.
@Module
@InstallIn(SingletonComponent::class)
object AnalyticsModule {

    @Singleton
    @Provides
    fun provideAnalyticsService(): AnalyticsService {
        return Retrofit.Builder()
            .baseUrl("https://example.com")
            .build()
            .create(AnalyticsService::class.java)
    }
}
```

Scoping and ViewModels

Originally if no scoping is done then activity retain a new instance upon each configuration change. Like this:

```
class ExampleActivity : AppCompatActivity() {
    private val analyticsAdapter = AnalyticsAdapter()
    ...
}
```

In Hilt this will look like this:

```
@ActivityScoped
class AnalyticsAdapter @Inject constructor() { ... }

@AndroidEntryPoint
class ExampleActivity : AppCompatActivity() {

    @Inject lateinit var analyticsAdapter: AnalyticsAdapter

}
```

The AnalyticsAdapter is scoped here to the Activity. When a new instance of ExampleActivity is created (e.g. the activity goes through a configuration change), a new instance of AnalyticsAdapter will be created.

To get the same instance we can achieve that through view models or with Hilt (with or without ViewModels)

With View Models

```
class AnalyticsAdapter() { ... }

class ExampleViewModel() : ViewModel() {
    val analyticsAdapter = AnalyticsAdapter()
}

class ExampleActivity : AppCompatActivity() {

    private val viewModel: ExampleViewModel by viewModels()
    private val analyticsAdapter = viewModel.analyticsAdapter

}
```

With Hilt (No ViewModels) we use **@ActivityRetainedScoped** that scope AnalyticsAdapter to the **ActivityRetainedComponent** which also survives **configuration changes**

```
@ActivityRetainedScoped
class AnalyticsAdapter @Inject constructor() { ... }

@AndroidEntryPoint
class ExampleActivity : AppCompatActivity() {

    @Inject lateinit var analyticsAdapter: AnalyticsAdapter
}
```

With Hilt and View Models - There is one major difference:

First, A Hilt View Model is a [Jetpack ViewModel](#) which his constructor injected by Hilt. To enable injection of a ViewModel by Hilt use the [@HiltViewModel](#) annotation:

```
@HiltViewModel
class ExampleViewModel @Inject constructor(
    val stateHandle: SavedStateHandle,
    val analyticsAdapter: AnalyticsAdapter
): ViewModel() { }
```

[SavedStateHandle](#) is a default binding available to all Hilt View Models (more on default bindings later on), while AnalyticsAdapter is a dependency which want to provide to the View Model. **This way of passing parameters to the view model is the preferred way over the Factory methods.**

But before we discuss this dependency scope let's look at how the activity or fragment retain an instance of that ViewModel.

The activity or fragments annotated with [@AndroidEntryPoint](#) can get the ViewModel instance as normal using ViewModelProvider or the by viewModels() KTX extension:

```
@AndroidEntryPoint
class ExampleActivity() : AppCompatActivity() {

    private val viewModel : ExampleViewModel by viewModels()
    private val analyticsAdapter = viewModel.analyticsAdapter
}
```

Only dependencies from the [ViewModelComponent](#) and its parent components

can be provided into the ViewModel.

All Hilt View Models are provided by the ViewModelComponent which follows the same lifecycle as a ViewModel, i.e. it survives configuration changes. To scope a dependency to a ViewModel use the [@ViewModelScoped](#) annotation.

If we own the class it will look like this:

```
@ViewModelScoped
class AnalyticsAdapter @Inject constructor() {}
```

If it is from a library then it will probably look like this:

```
@Module
@InstallIn(ViewModelComponent::class)
object AnalyticsModule {

    @ViewModelScoped
    @Provides
    fun provideAnalyticsAdapter() = AnalyticsAdapter()
}
```

A `@ViewModelScoped` type will make it so that a single instance of the scoped type is provided across all dependencies injected into the Hilt View Model.

Other instances of a ViewModel that requests the scoped instance will receive a different instance.

If a single instance needs to be shared across various View Models then it should be scoped using either `@ActivityRetainedScoped` or `@Singleton`.

For example, we can scope a dependency to be shared within a single ViewModel as such:


```
@Module
@InstallIn(ViewModelComponent::class)
internal object ViewModelMovieModule {
    @Provides
    @ViewModelScoped
    fun provideRepo(handle: SavedStateHandle) =
        MovieRepository(handle.getString("movie-id"));
}

class MovieDetailFetcher @Inject constructor(
    val movieRepo: MovieRepository
)

class MoviePosterFetcher @Inject constructor(
    val movieRepo: MovieRepository
)

@HiltViewModel
class MovieViewModel @Inject constructor(
    val detailFetcher: MovieDetailFetcher,
    val posterFetcher: MoviePosterFetcher
) : ViewModel {
    init {
        // Both detailFetcher and posterFetcher will contain the same instance of
        // the MovieRepository.
    }
}
```

Or another example:

```
@ViewModelScoped // Scopes type to the ViewModel
class UserInputAuthData(
    private val handle: SavedStateHandle // Default binding in ViewModelComponent
) { /* Cached data and logic here */ }

class RegistrationViewModel(
    private val userInputAuthData: UserInputAuthData,
    private val validateUsernameUseCase: ValidateUsernameUseCase,
    private val validatePasswordUseCase: ValidatePasswordUseCase
) : ViewModel() { /* ... */ }

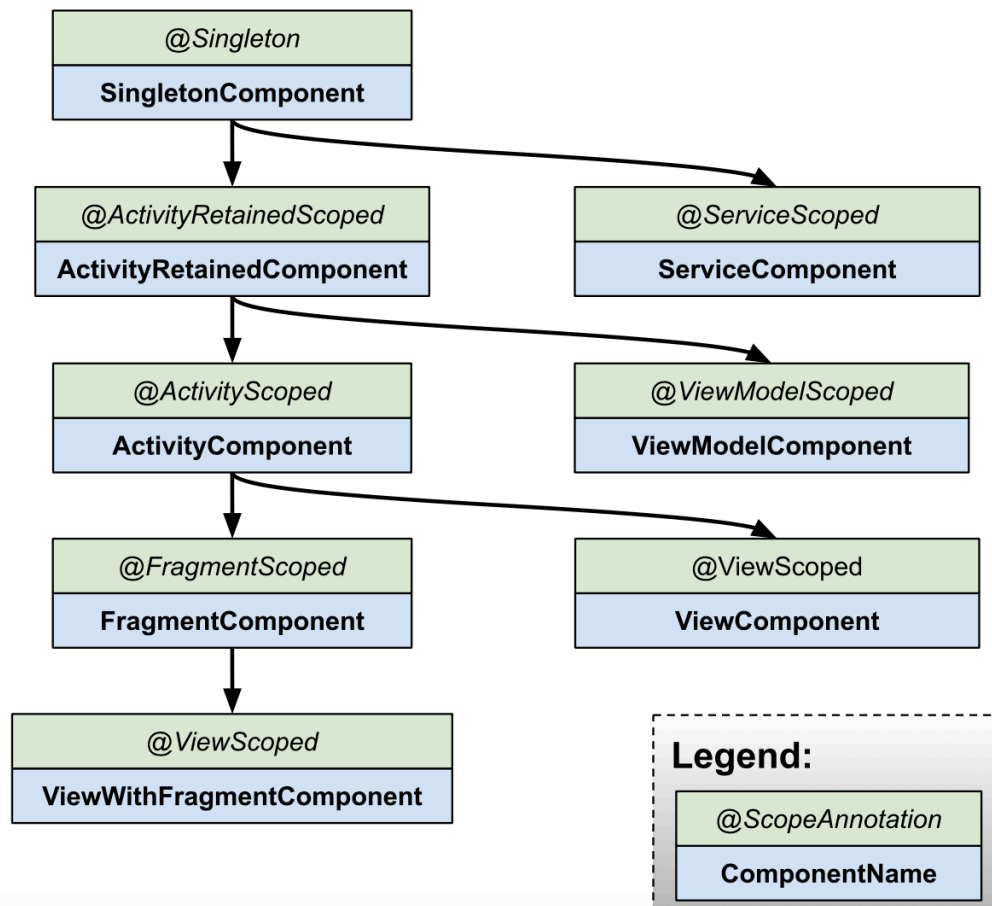
class LoginViewModel(
    private val userInputAuthData: UserInputAuthData,
    private val validateUsernameUseCase: ValidateUsernameUseCase,
    private val validatePasswordUseCase: ValidatePasswordUseCase
) : ViewModel() { /* ... */ }

class ValidateUsernameUseCase(
    private val userInputAuthData: UserInputAuthData,
    private val repository: UserRepository
) { /* ... */ }

class ValidatePasswordUseCase(
    private val userInputAuthData: UserInputAuthData,
    private val repository: UserRepository
) { /* ... */ }
```

Since `UserInputAuthData` is scoped to the `ViewModel`, `RegistrationViewModel` and `LoginViewModel` will receive a *different instance* of `UserInputAuthData`. However, the `UseCase` dependencies of each `ViewModel` use the *same instance* that its `ViewModel` uses.

Installing a module into a component allows its bindings to be accessed as a dependency of other bindings in that component or in any child component below it in the component hierarchy:



Predefined qualifiers in Hilt

Hilt provides some predefined qualifiers. For example, as you might need the Context class from either the application or the activity, Hilt provides the @ApplicationContext and @ApplicationContext qualifiers.

```

class AnalyticsServiceImpl @Inject constructor(
    @ApplicationContext context: Context
) : AnalyticsService { ... }

// The Application binding is available without qualifiers.
class AnalyticsServiceImpl @Inject constructor(
    application: Application
) : AnalyticsService { ... }
  
```

```
class AnalyticsAdapter @Inject constructor(
    @ApplicationContext context: Context
) { ... }

// The Activity binding is available without qualifiers.
class AnalyticsAdapter @Inject constructor(
    activity: FragmentActivity
) { ... }
```

This is because Each Hilt component comes with a set of default bindings that Hilt can inject as dependencies into your own custom bindings.

Android component	Default bindings
SingletonComponent	Application
ActivityRetainedComponent	Application
ViewModelComponent	SavedStateHandle
ActivityComponent	Application, Activity
FragmentComponent	Application, Activity, Fragment
ViewComponent	Application, Activity, View
ViewWithFragmentComponent	Application, Activity, Fragment, View
ServiceComponent	Application, Service

Integration with the Jetpack navigation library

Add the following additional dependencies to your app Gradle file:

```
implementation("androidx.hilt:hilt-navigation-fragment:1.0.0")
```

If your ViewModel is **scoped to the navigation graph**, use the `hiltNavGraphViewModels` function that works with fragments that are annotated with `@AndroidEntryPoint`.

```
val viewModel: ExampleViewModel by hiltNavGraphViewModels(R.id.my_graph)
```

See a nice example <https://stackoverflow.com/questions/66497047/hilt-doesnt-inject-a-scoped-viewmodel>

Inject dependencies in classes not supported by Hilt

Hilt comes with support for the most common Android classes. However, you might need to perform field injection in classes that Hilt doesn't support.

In those cases, you can create an entry point using the `@EntryPoint` annotation. An entry point is the boundary between code that is managed by Hilt and code that is not. It is the point where code first enters into the graph of objects that Hilt manages. Entry points allow Hilt to use code that Hilt does not manage to provide dependencies within the dependency graph.

For example, Hilt doesn't directly support `content providers`. If you want a content provider to use Hilt to get some dependencies, you need to define an interface that is annotated with `@EntryPoint` for each binding type that you want and include qualifiers. Then add `@InstallIn` to specify the component in which to install the entry point as follows:

```
class ExampleContentProvider : ContentProvider() {  
  
    @EntryPoint  
    @InstallIn(SingletonComponent::class)  
    interface ExampleContentProviderEntryPoint {  
        fun analyticsService(): AnalyticsService  
    }  
  
    ...  
}
```

To access an entry point, use the appropriate static method from `EntryPointAccessors`. The parameter should be either the component instance or the `@AndroidEntryPoint` object that acts as the component holder. Make sure that the component you pass as a parameter and the `EntryPointAccessors` static method both match the Android class in the `@InstallIn` annotation on the `@EntryPoint` interface:

```
class ExampleContentProvider: ContentProvider() {
    ...

    override fun query(...): Cursor {
        val appContext = context?.applicationContext ?: throw IllegalStateException()
        val hiltEntryPoint =
            EntryPointAccessors.fromApplication(appContext, ExampleContentProviderEntryPoint::class.java)

        val analyticsService = hiltEntryPoint.analyticsService()
        ...
    }
}
```

In this example, you must use the `ApplicationContext` to retrieve the entry point because the entry point is installed in `SingletonComponent`. If the binding that you wanted to retrieve were in the `ActivityComponent`, you would instead use the `ActivityContext`.

Unit tests

Hilt isn't necessary for unit tests, since when testing a class that uses constructor injection, you don't need to use Hilt to instantiate that class. Instead, you can directly call a class constructor by passing in fake or mock dependencies, just as you would if the constructor wasn't annotated:

```
@ActivityScoped
class AnalyticsAdapter @Inject constructor(
    private val service: AnalyticsService
) { ... }

class AnalyticsAdapterTest {

    @Test
    fun `Happy path`() {
        // You don't need Hilt to create an instance of AnalyticsAdapter.
        // You can pass a fake or mock AnalyticsService.
        val adapter = AnalyticsAdapter(fakeAnalyticsService)
        assertEquals(...)
    }
}
```

Hilt testing guide

One of the benefits of using dependency injection frameworks like Hilt is that it makes testing your code easier.

You can read more about testing with hilt here

<https://developer.android.com/training/dependency-injection/hilt-testing>

<https://developer.android.com/training/dependency-injection>

<https://developer.android.com/training/dependency-injection/hilt-android>

Full usage example of App Architecture, Retrofit, Room, Hilt and Coroutines.

Download the Starter Files for this project here

<https://drive.google.com/file/d/1yM2Y3lc4S20mAoiCWNKrAisoT-Jg07q4/view?usp=sharing>

Download the starter Project from here (Optional - not necessary for this tutorial):

<https://drive.google.com/file/d/1i0wSsU6s9mqnaePDqYl8Ls0MfB6GUBCS/view?usp=sharing>

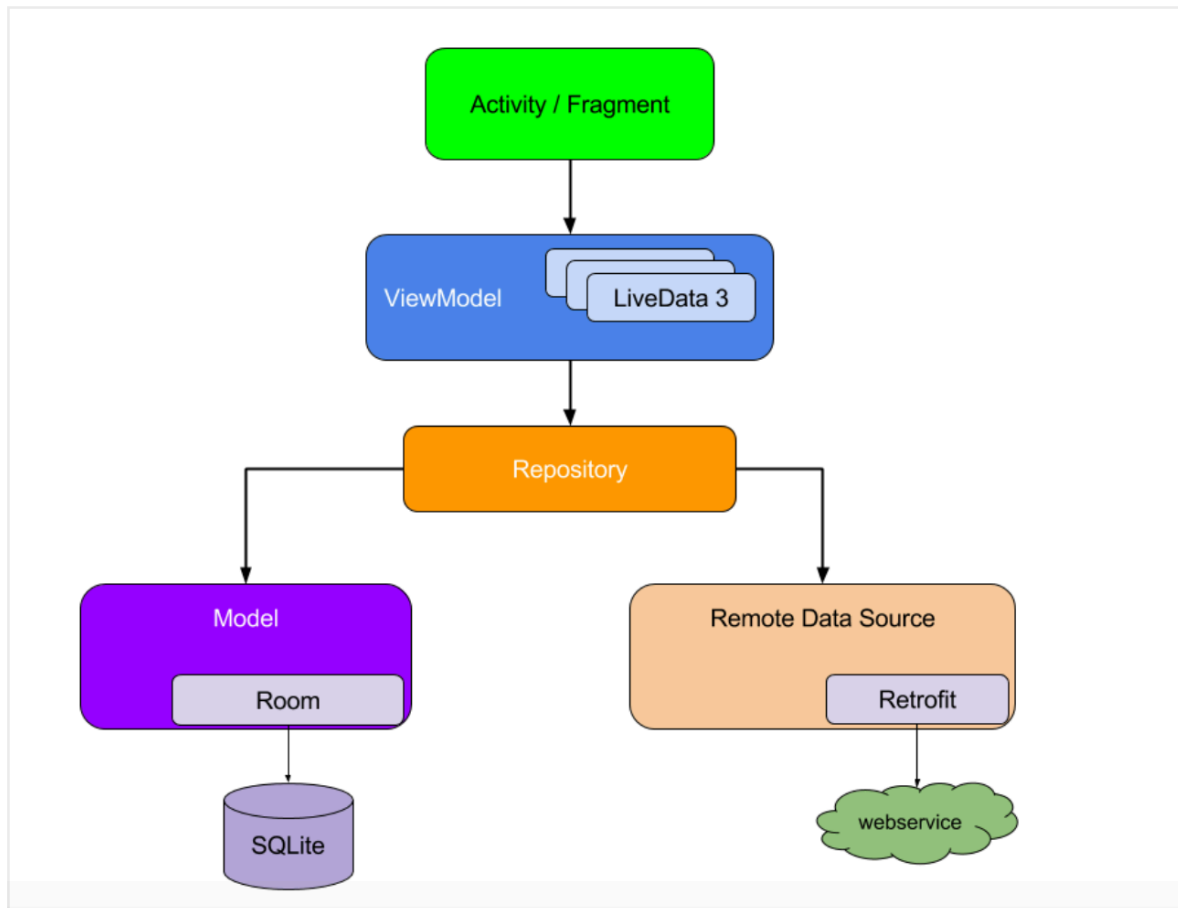
Download the Full App Created in this Guide:

https://drive.google.com/file/d/1sl_T-_zr8w0z7riMyvpNlReB8WcYN5N6/view?usp=sharing

Download the Generic useful classes to use in other projects:

https://drive.google.com/file/d/1fpH2qcyI0020pY0ZbB1yN69wmU0xT_NG/view?usp=sharing

We are going to put it all together now for building a clean architecture app that will communicate with both local and remote databases and will use Hilt dependency injection library for reducing boilerplate code.



Before we begin diving into our code let's look at the remote database. We will use one of the most Common web service for testing: the rick and morty API. You can explore it here <https://rickandmortyapi.com> just visit the docs and look the REST API, we have the characters, locations and episodes data, we will get all the characters and show them nicely in a RecyclerView and by clicking on each character we will show it in details.

So our base url will be:

<https://rickandmortyapi.com/api>

And we will access the /character resource

Lets look at the returned JSON:

You can use this site for json formatting - just paste the full url

<https://jsonformatter.curiousconcept.com>

<https://rickandmortyapi.com/api/character>


```
{
  "info": { },
  "results": [ ]
}
```

```
{
  "info": {
    "count": 671,
    "pages": 34,
    "next": "https://rickandmortyapi.com/api/character?page=2",
    "prev": null
  },
  "results": [
    {
      "id": 1,
      "name": "Rick Sanchez",
      "status": "Alive",
      "species": "Human",
      "type": "",
      "gender": "Male",
      "origin": {
        "name": "Earth (C-137)",
        "url": "https://rickandmortyapi.com/api/location/1"
      },
      "location": {
        "name": "Earth (Replacement Dimension)",
        "url": "https://rickandmortyapi.com/api/location/20"
      },
      "image": "https://rickandmortyapi.com/api/character/avatar/1.jpeg",

```

```
"results": [
  {
    "id": 1,
    "name": "Rick Sanchez",
    "status": "Alive",
    "species": "Human",
    "type": "",
    "gender": "Male",
    "origin": { },
    "location": { },
    "image": "https://rickandmortyapi.com/api/character/avatar/1.jpeg",
    "episode": [ ],
    "url": "https://rickandmortyapi.com/api/character/1",
    "created": "2017-11-04T18:48:46.250Z"
  },

```

Even though we need only the JSON objects in the result array in order to simplify the Gson Builder factory we will create data classes for the character, the info object and the root object which will contain a list of characters and the

total info.

Step 1 - Project Setup

Go ahead open a new Android Studio Project and create the following project structure:

In our Project we will divide our project to:

data - it will include our models, local and remote database data operations and repositories.

di - it will include all of our injected dependencies and we will to that with the help of Hilt.

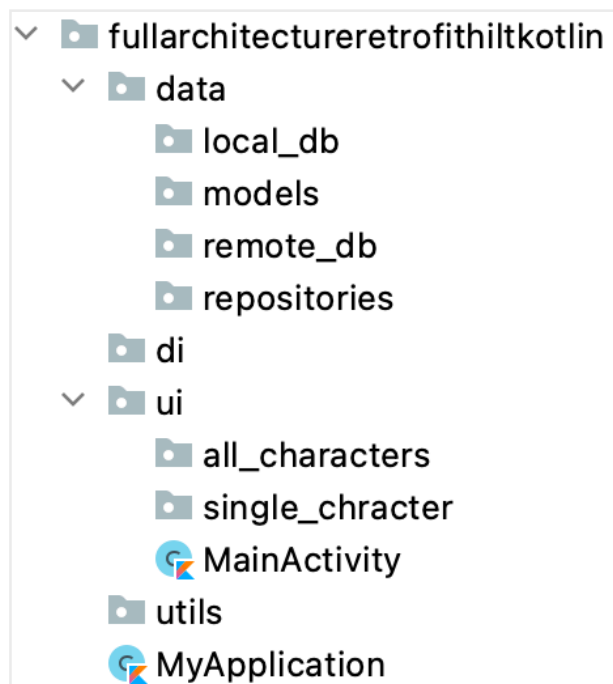
ui - all of our UI related components and their ViewModels (yes view models goes there).

utils - all of the helper classes and project related general functions.

Go ahead and create the packages mentioned above and their sub folders.

Move your MainActivity to the ui package and in the root package create your Application class for Hilt and add it's name to your manifest file.

In the end it should look like this:



```
<application
    android:name=".MyApplication"
```

Also in the utils package a create Constants class to hold out Base url on which we will add the specific resource for each GET call (remember to add the / at the end of the path:

```
class Constants {  
  
    companion object {  
        const val BASE_URL = "https://rickandmortyapi.com/api/"  
    }  
}
```

If you are already in your Manifest don't forget to add the Internet install time permission used for our retrofit calls:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Now let's go to our project and app Gradle files and get all the 3rd party libraries dependencies:

In your project Gradle file we just need to add Hilt:

```
dependencies {
```

```
    ...  
    classpath 'com.google.dagger:hilt-android-gradle-plugin:2.38.1'  
}
```

In your app Gradle file we need add the plugins:

```
id 'kotlin-kapt'  
id 'dagger.hilt.android.plugin'
```

And a whole bunch of stuff under our dependency :

```
//Retrofit
```

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'  
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

```
//Lifecycle
```

```
def lifecycle_version = "2.3.1"  
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"  
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"  
implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"  
implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
```

```
//Kotlin Coroutines
```

```
def coroutines_android_version = '1.5.2'  
implementation "org.jetbrains.kotlin:kotlinx-coroutines-core:  
$coroutines_android_version"  
implementation "org.jetbrains.kotlin:kotlinx-coroutines-android:  
$coroutines_android_version"
```

```
//Hilt
implementation 'com.google.dagger:hilt-android:2.38.1'
implementation "androidx.hilt:hilt-lifecycle-viewmodel:1.0.0-alpha03"
kapt 'com.google.dagger:hilt-android-compiler:2.38.1'
kapt "androidx.hilt:hilt-compiler:1.0.0"

//Room
def room_version = "2.3.0"
implementation "androidx.room:room-runtime:$room_version"
implementation "androidx.room:room-ktx:$room_version"
kapt "androidx.room:room-compiler:$room_version"

//Navigation
def nav_version = "2.3.5"
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"

//Glide
implementation 'com.github.bumptech.glide:glide:4.12.0'
kapt 'com.github.bumptech.glide:compiler:4.12.0'
```

Go ahead now sync your project

Step 2 - Create your models

Create all the data Classes we have talked about in the beginning. When you create your models do it carefully and look at the Json response. You don't have to include properties for each JSON field in the response but why not, maybe we will need it, if you are lazy we only need the **id, name, gender, status and species** but make sure you use the exact same names for the properties as in the response or retrofit won't be able to create your objects.

If for some reason you want a different name you can use `@SerializedName([JSON field name])`- same as `@ColumnInfo` we have seen in Room.

```
@SerializedName( value: "image")
val picture : String,
```

One last thing pre-plan your room annotations in your Data classes

In the end it should look like this (for this project the classes don't have to be data class but it is wiser):

```
@Entity(tableName = "characters")
data class Character(
    @PrimaryKey
    val id : Int,
    val name : String,
    val status : String,
    val species : String,
    val type : String,
    val gender : String,
    val image : String,
    val url: String,
    val created: String
) {
}
```

```
data class Info(
    val count : Int,
    val pages : Int,
    val next : String,
    val prev : Any
) {
}
```

```
data class AllCharacters(
    val info : Info,
    val results : List<Character>
) {
}
```

Step 3 - Adding Hilt

Let's start with Hilt. Why do we need Hilt you can ask but think of all the dependencies we have here. The view Model will need the repository, the repository will require both the local and the remote database. Each database requires his service and so on. This is just a simple app, the more complex it gets the more dependencies it has.

This is where Hilt comes to rescue, we can auto inject each dependency and not have to worry about writing unnecessary code. We just need to tell Hilt where we need a dependency and where to get it from, it will connect the dots and take care of the object creation and their lifecycles.

First add the **@HiltAndroidApp** to your application file

```
@HiltAndroidApp
class MyApplication() : Application() {
}
```

Create both of your fragments and add to them **@AndroidEntryPoint** annotation meaning they can get the Hilt dependencies.

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
}
```

```
@AndroidEntryPoint
class SingleCharacterFragment : Fragment() {
}
```

```
@AndroidEntryPoint
class AllCharactersFragment : Fragment() {
}
```

Also create their View Models and add the **@HiltViewModel** and the **@Inject** constructor

```
@HiltViewModel
class AllCharactersViewModel @Inject constructor() : ViewModel() {
}
```

```
@HiltViewModel
class SingleCharacterViewModel @Inject constructor() : ViewModel() {
}
```

Now since we don't own the retrofit classes and we can't create an injected constructor we must build the Retrofit module inside our di package. There we

must annotate by @Provides each of the dependencies we need to provide, this will be our bag of dependencies. The class will be installed in the SingletonComponent, meaning it will be available for all of the app. We also use the @Singleton annotation in cases where we want only one instance of the provided dependency. For now we just add a function that returns a single instance of the retrofit and also his constructor's Gson dependency.

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    @Singleton
    fun provideRetrofit(gson: Gson) : Retrofit {
        return Retrofit.Builder().baseUrl(Constants.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create(gson)).build()
    }

    @Provides
    fun provideGson() : Gson = GsonBuilder().create()
}
```

One more thing. I want to explain more about the Gson. Here we are not doing any specific deserialization meaning our response JSON object will be directly mapped into the corresponding object - the json object has two keys one for the info and one for the result and so does the Kotlin data class. But if we wanted to map the result to a character class without the extra classes on the way then we would have needed to go into the root object get the array under the key results and fit each json object there to the character class - this is custom deserialization. For this we would have need to pass a custom Gson converter factory and not the standard one like we did here.

A nice and very simple tutorial on how to do this can be found here:

<https://www.woolha.com/tutorials/retrofit-2-define-custom-gson-converter-factory>

We will get back to this module and add provider functions to the rest it's of the data parts. But now let's go ahead and create them, only then we can supply dependency for them.

Step 4 - Room Database

We will start From the easy part - Room Database

Besides the retrofit work, we are also interested in storing and getting data from a local database. We need it in order to show at least something to our users when they are out of connection or before the data is fetched - with slow connection. We are going to use it as a *cache* for our system.

Add your Dao and create functions to retrieve a character by id and all characters, and functions to insert a character or character list. Please make the insert functions suspended and the only thing we need is to execute it from a coroutine scope. Room will take care of their implementation including their background capabilities. Please note that the fetching functions don't have to be suspended because LiveData is already asynchronous - it is working on the IO Dispatchers.

Our Dao should look like this:

```
@Dao
interface CharacterDao {

    @Query(value: "SELECT * FROM characters")
    fun getAllCharacters() : LiveData<List<Character>>

    @Query(value: "SELECT * FROM characters WHERE id = :id")
    fun getCharacter(id : Int) : LiveData<Character>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCharacter(character: Character)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCharacters(characters : List<Character>)

}
```

And our app database will look like this


```
@Database(entities = [Character::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {

    abstract fun characterDao() : CharacterDao

    companion object {

        @Volatile
        private var instance : AppDatabase? = null

        fun getDatabase(context: Context) : AppDatabase =
            instance ?: synchronized(lock: this) {
                Room.databaseBuilder(context.applicationContext, AppDatabase::class.java, name: "characters")
                    .fallbackToDestructiveMigration().build().also { it: AppDatabase
                        instance = it
                    }
            }
    }
}
```

Please note:

fallbackToDestructiveMigration() - tells Room that if the database version had changed and No Migration guide is found not to throw an exception but instead delete the old table and recreate it.

Step 5 - Retrofit calls

We are all done with the caching of the data and move to fetching. Let's create our data fetching service (same as the Dao in Room). This interface will be called CharacterService and it will contain two functions: both annotated with @Get("[path]"), both suspended meaning they can be executed on a background thread and both return the retrofit Response object with the Kotlin class that that json object should be parsed to. Amazing what could be achieved in a single line of code!

The code should look like this:

```
interface CharacterService {

    @GET(value: "character")
    suspend fun getAllCharacters() : Response<AllCharacters>

    @GET(value: "character/{id}")
    suspend fun getCharacter(@Path(value: "id") id: Int) : Response<Character>
}
```

Note:

You can use The @Query annotation to add the functions parameters and append them as the Query parameters.

For example:

If we want to add to the base URL this path `"/maps/api/geocode/json?"` With that parameters: `"address=90210&sesnsor=false"`

We will create the following Get call

```
@GET("/maps/api/geocode/json?")
```

```
suspend fun getLocationInfo(@Query("address") String
zipCode,@Query("sensor") boolean sensor);
```

Retrofit also offers a returned Call that can be executed async but here the function itself is suspended so we don't need to use retrofit's background execution queue and execute Call on it.

The functions returns Response. When we invoke it we can check if it is successful and it has a data that can be null for both cases and a message that won't be null if there was an error and in that case there is also his code. More then that some exceptions can occur while executing the retrofit call (meaning not getting the response at all).

It seems like allot of different branching when all we care about is Success, Failure or Loading to be checked in a nicely organize when() {} clause. And we are going to do allot of work to achieve that, but this work will done once for all Requests. It will be Generic enough to serve us in every retrofit invocation.

First create your Wrapper class for the Response.

The base logic is to create a sealed Status object that works on a generic covariance that will be its data. From this sealed class there will be three derived classes: Success, Error (which will hold an additional error message) and Loading. We will create the Resource class with Status as her property and three Factory methods which we will create the suitable Response object with each status option :

```
class Resource<out T> private constructor(val status: Status<T>) {
    companion object {
        fun<T> success(data : T) = Resource(Success(data))
        fun<T> error(message: String, data : T? = null) = Resource(Error(message,data))
        fun<T> loading(data : T? = null) = Resource>Loading(data))
    }
}

sealed class Status<out T>(val data: T? = null)

class Success<T>(data: T) : Status<T>(data)
class Error<T>(val message:String, data: T? = null) : Status<T>(data)
class Loading<T>(data: T? = null) : Status<T>(data)
```

When we get an instance of this Resource wrapper class it will be very easy to check:

```
when (it.status) {  
    is Success -> {//do something with the data that is not null  
  
    }  
  
    is Error -> {//read the error message (data may be null)  
  
    }  
  
    is Loading -> {//wait (data may be null)  
  
    }  
}
```

More than that, because Status is sealed the compiler will warn us if we forget to check any of it's subclasses.

So the wrapper class is ready now we need to actually call the function and wrap the Response with that Resource.

To do this in A Generic way we will create a base class with getResult function that will get the Response and return the appropriate Resource. Then we will inherit from that generic class to a specific data source for our service and call the generic getResult with functions from our service. Our data source will have an @injected constructor that will get the specific service and create wrapper functions for each of the service function. Don't worry about the injection we will provide the injected service in our AppModule

While we created our Resource classes in the util package generate this two classes in the remote db folder.

our new added code will look like this:

```

abstract class BaseDataSource {

    protected suspend fun <T> getResult(call : suspend () -> Response<T>) : Resource<T> {

        try {
            val result = call() //we invoke the given suspended function(we are suspended also)
            if(result.isSuccessful) {
                val body = result.body()
                if(body!=null) return Resource.success(body)
            }
            return Resource.error( message: "Network call has failed for a following reason: " +
                "${result.message()} ${result.code()}")
        }catch (e : Exception) {
            return Resource.error( message: "Network call has failed for a following reason: " +
                (e.localizedMessage ?: e.toString())
            )
        }
    }
}

```

```

@Singleton
class CharacterRemoteDataSource @Inject constructor(
    private val characterService: CharacterService) : BaseDataSource() {

    suspend fun getCharacters() = getResult { characterService.getAllCharacters() }
    suspend fun getCharacter(id : Int) = getResult { characterService.getCharacter(id) }
}

```

Step 6 - Repository

The very last thing we have to do in terms of our data is to create the Repository.

First let's explain our policy for local and remote data fetching:

- First we need to let our LiveData know that we are looking for the Character, so that should be the LOADING state.
- Then, we would like to get that character from the local data source, because it is faster than getting it from the internet. If it finds it, we are changing the state to a SUCCESS
- Regardless of the result of the local database operation, we would want to keep our app synched, so we are fetching the characters from the internet as well (but remember that the ui thread won't be blocked and the user can already see the correct characters information).
- Finally, we need to save our result from the remote call in the local database, in order to keep it updated.

To achieve that we can use a Generic get function that receives three functions as parameters: One for local fetching, one for remote fetching and one for saving data. Then we will use the LiveData coroutine builder to create scope for

running our suspended functions synchronously and get back the result as a LiveData object (please note that we need to tell the LiveData builder we want to run our job on the Dispatchers.IO because this default LiveData scope is the Main UI thread where emit is called). We are creating a scope where the suspended functions can wait for each other although they run in a background thread, and updating the data they fetched using emit which is called on the Main UI thread for anyone who is observing these returned LiveData. You can also emit multiple values from the block. Each emit() call suspends the execution of the block until the LiveData value is set on the main thread. You can read more here:

<https://developer.android.com/topic/libraries/architecture/coroutines#livedata>

We will use both emit() and emitSource() function from within the LiveData scope. These functions defined by the LiveDataScope interface and are used to update the LiveData value or it's source. Meaning emit will be used to update the LiveData stored value and emitSource the LiveData itself and then each change will be auto updates by the LiveData. These are both suspended functions meaning it will pause the scope until the LiveData is updated. Remember: Inside a suspended function, calls to other suspended functions behave like normal function calls. We stop and wait. We work on the same coroutine that can be stoped and continued.

So our helper function will look like that (put it a general DataFunction Kotlin file located in the util package):

If you are wondering why you see two generics it is because we need to distinguish the value stored in the LiveData from the value returned by the call, for example in a single function call we are getting all the characters from the remote db we get a Response<AllCharacters> while from the local db we get LiveData<List<Character>>. The A represent the retrofit's generic while the T is the Room generic.

```
fun <T, A> performFetchingAndSaving(localDbFetch: () -> LiveData<T>,
    remoteDbFetch: suspend () -> Resource<A>,
    localDbSave: suspend (A) -> Unit) : LiveData<Resource<T>> =

    LiveData(Dispatchers.IO) { this: LiveDataScope<Resource<T>>
        emit(Resource.Loading()) //this will tell the live data we are loading

        val source = localDbFetch().map { Resource.success(it) } //mapping the returned
        //live data his wrapper class
        emitSource(source) //setting the new source for live data

        val fetchResource = remoteDbFetch()

        if(fetchResource.status is Success)
            localDbSave(fetchResource.status.data!!) //saving the data will also update vi any change of data
            //because we used emitSource with the local db livedata
            //since it is LiveData
        else if(fetchResource.status is Error)
        {
            emit(Resource.error(fetchResource.status.message))
            emitSource(source) //because each call to emit remove the previous value
            //we want to last valid value to be stored even if it is empty
        }
    }
}
```

And last but not least we will write the Character Repository. It's @Injected constructor will get the local and the remote services by Hilt and will use this function above to do all the work. Please note that we use the @Singleton scope so one instance of the repository is for all of the app. This repository will be auto created and injected later on to our view model.

```
@Singleton
class CharacterRepository @Inject constructor(
    private val remoteDataSource : CharacterRemoteDataSource,
    private val localDataSource : CharacterDao
){

    fun getCharacters() = performFetchingAndSaving(
        {localDataSource.getAllCharacters()},
        {remoteDataSource.getCharacters()},
        {localDataSource.insertCharacters(it.results)}
    )

    fun getCharacter(id : Int) = performFetchingAndSaving(
        {localDataSource.getCharacter(id)},
        {remoteDataSource.getCharacter(id)},
        {localDataSource.insertCharacter(it)}
    )
}
```

Add put them all together in you AppModule dependency bag

Remember if you @provide a class then all of its constructor parameters also have to be provided!

Our final AppModule should look like that:

```

@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    fun provideGson() : Gson = GsonBuilder().create()

    @Provides
    @Singleton
    fun provideRetrofit(gson : Gson) : Retrofit {
        return Retrofit.Builder().baseUrl(Constants.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create(gson)).build()
    }

    @Provides
    fun provideCharacterService(retrofit: Retrofit) : CharacterService =
        retrofit.create(CharacterService::class.java)

    @Provides
    @Singleton
    fun provideLocalDataBase(@ApplicationContext appContext : Context) : AppDatabase =
        AppDatabase.getDatabase(appContext)

    @Provides
    @Singleton
    fun provideCharacterDao(database: AppDatabase) = database.characterDao()
}

```

When we want to create our Retrofit service to execute our queries we can call create on our retrofit instance. It will create an implementation of the API endpoints defined by the service interface.

Please also note that we are using **@ApplicationContext** that allows hilt to provide application context without having to explicitly specify how to obtain it. And also note that we don't need to provide the CharacterRepository and the CharacterRemoteDataSource since we they have the @Inject constructor meaning Hilt can generate these classes without the need to explicitly tell him how.

Thats It for our data and the dependencies! Now all we have to do is the easy part UI!

Step 7 - UI

First copy the four xml files found on the starter into your res/layout folder Please note a few things:

1. In the activity_main we have a custom tool bar this for setup with the navigation component
2. Still in activity_main We Have the FragmentContainerView please notice his id and the nav graph id - when you create the nav graph use this id or change it here to what you will use

Go ahead and add your navigation graph to your resources with name corresponding to the Container mentioned above. Add both of your fragments and create an action between them like here:



Add your view binding to the app Gradle file

```
viewBinding {
    enabled = true
}
```

Define your view binding and pass the root view.

Get your Navigation controller And connect you toolbar to your navigation component for the purpose of showing the current fragment label in the app bar and navigating back. Your MainActivity should look like that:

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        val navHostFragment = supportFragmentManager.findFragmentById(R.id.nav_host_fragment)
            as NavHostFragment
        val navController = navHostFragment.navController
        val appBarConfiguration = AppBarConfiguration(navController.graph)
        binding.toolbar.setupWithNavController(navController, appBarConfiguration)
    }
}
```

Let's start with all the characters and their RecyclerView adapter - use Glide for the pictures and also create an interface to pass the item click to the fragment who will implement the interface. When clicked pass the character id. In this example I prefer to get the listener in the adapters constructor and only it. The list of characters will be updated through setCharacters function you will add to the adapter.

Here is the full Adapter code:

```
class CharactersAdapter(private val listener : CharacterItemClickListener) :
    RecyclerView.Adapter<CharactersAdapter.CharacterViewHolder>() {

    private val characters = ArrayList<Character>()

    class CharacterViewHolder(private val itemBinding: ItemCharacterBinding,
                             private val listener: CharacterItemClickListener)
        : RecyclerView.ViewHolder(itemBinding.root),
        View.OnClickListener {

        private lateinit var character: Character

        init {
            itemBinding.root.setOnClickListener(this)
        }

        fun bind(item: Character) {
            this.character = item
            itemBinding.name.text = item.name
            itemBinding.speciesAndStatus.text = "${item.species} - ${item.status}"
            Glide.with(itemBinding.root)
                .load(item.picture)
                .circleCrop()
                .into(itemBinding.image)
        }

        override fun onClick(v: View?) {
            listener.onCharacterClick(character.id)
        }
    }
}
```

```
fun setCharacters(characters : Collection<Character>) {
    this.characters.clear()
    this.characters.addAll(characters)
    notifyDataSetChanged()
}

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CharacterViewHolder {
    val binding = ItemCharacterBinding.inflate(LayoutInflater.from(parent.context), parent, attachToParent: false)
    return CharacterViewHolder(binding, listener)
}

override fun onBindViewHolder(holder: CharacterViewHolder, position: Int) =
    holder.bind(characters[position])

override fun getItemCount() = characters.size

interface CharacterItemClickListener {
    fun onCharacterClick(characterId : Int)
}
```

Now for the final stage: Your Fragments and their View Models!

Let's start with AllCharacters - The ViewModel should supply the list of characters. All it needs in its Injected constructor is the repository. We will create a single characters property and get it from the repository. This will be

the observable LiveData.

```
@HiltViewModel
class AllCharactersViewModel @Inject constructor(
    characterRepository: CharacterRepository) : ViewModel(){

    val characters = characterRepository.getCharacters()
}
```

In the Fragment get your view bindings. Don't forget that the fragment outlives its views so release your binding in the onDestroyView. Don't worry about the hassle, in the end you will get a property delegate that does this automatically while observing the corresponding fragment's lifecycle events.

Create the adapter and implement his Listener, the item click should perform the navigation's one and only pre-made action and pass a bundle containing the supplied character id.

Get the RecyclerView set its layout manager and the created Adapter above. Now observe the characters from your view model and upon invocation check your status. in case of loading show the progress bar, In case of Success hide it set the adapter's characters, and in case of an error hide it and prompt the error message in a Toast.

```
@AndroidEntryPoint
class AllCharactersFragment : Fragment(), CharactersAdapter.CharacterItemListener {

    private val viewModel : AllCharactersViewModel by viewModels()

    private var _binding : CharactersFragmentBinding? = null
    //private val binding : CharactersFragmentBinding by autoCleared()
    private val binding get() = _binding!!

    private lateinit var adapter: CharactersAdapter

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = CharactersFragmentBinding.inflate(inflater, container, attachToParent: false)

        return binding.root
    }
}
```

```

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    adapter = CharactersAdapter( listener: this)
    binding.charactersRv.layoutManager = LinearLayoutManager(requireContext())
    binding.charactersRv.adapter = adapter

    viewModel.characters.observe(viewLifecycleOwner) { it: Resource<List<Character>>!
        when(it.status) {
            is Loading -> binding.progressBar.visibility = View.VISIBLE

            is Success -> {
                binding.progressBar.visibility = View.GONE
                adapter.setCharacters(ArrayList(it.status.data))
            }

            is Error -> {
                binding.progressBar.visibility = View.GONE
                Toast.makeText(requireContext(), it.status.message, Toast.LENGTH_SHORT).show()
            }
        }
    }
}
}

```

```

override fun onDestroyView() {
    super.onDestroyView()
    binding = null
}

override fun onCharacterClick(characterId: Int) {

    findNavController().navigate(R.id.action_allCharactersFragment_to_singleCharacterFragment,
        bundleOf( ...pairs: "id" to characterId))
}

```

Our very last part of this very long journey is the detailed character fragment and it's View Model.

First, our View model should get the repository and get a character by it's id. Now this is a tricky part. Think a little about how to solve this.

First we need a character property so we can observe it. But what will trigger the event?

We will create a character look up as a transformation of the id.

The id will have a public set function that we will invoke from the fragment.

The character field will be defined as a transformation of the id. As long as your app has an active observer associated with the character field, the field's value is recalculated and retrieved whenever id changes.

```

@HiltViewModel
class SingleCharacterViewModel @Inject constructor(
    private val characterRepository: CharacterRepository) : ViewModel() {

    private val _id = MutableLiveData<Int>()

    private val _character = _id.switchMap { it: Int!
        | characterRepository.getCharacter(it)
    }

    fun setId(id : Int) {
        _id.value = id
    }
}

```

There is a reason why we use the internal `_character` - it is a mutable live data and therefore can be dangerous to expose that is why we will return only LiveData with a public character

```
val character : LiveData<Resource<Character>> = _character
```

For more reading on transformations:

https://developer.android.com/topic/libraries/architecture/livedata#transform_livedata

And that's it for the View Model

As for the fragment do the same binding as before and when your view is created get the id from the arguments, remember to deal nicely with null, and set the value in the view model value, this will trigger your character observer and upon invocation will update the ui!

```

@AndroidEntryPoint
class SingleCharacterFragment : Fragment() {

    private val viewModel : SingleCharacterViewModel by viewModels()

    private var _binding : CharacterDetailFragmentBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = CharacterDetailFragmentBinding.inflate(inflater, container, attachToParent: false)
        return binding.root
    }
}

```

```

override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)

    viewModel.character.observe(viewLifecycleOwner) { it: Resource<Character>!
        when(it.status) {
            is Success -> {
                binding.progressBar.visibility = View.GONE
                updateCharacter(it.status.data!!)
                binding.characterCl.visibility = View.VISIBLE
            }
            is Loading -> {
                binding.progressBar.visibility = View.VISIBLE
                binding.characterCl.visibility = View.GONE
            }
            is Error -> {
                binding.progressBar.visibility = View.GONE
                Toast.makeText(requireContext(), it.status.message, Toast.LENGTH_SHORT).show()
            }
        }
    }

    arguments?.getInt( key: "id")?.let { it: Int
        viewModel.setId(it)
    }
}

```

```

fun updateCharacter(character:Character) {
    binding.name.text = character.name
    binding.gender.text = character.gender
    binding.species.text = character.species
    binding.status.text = character.status
    Glide.with(requireContext()).load(character.image).transform(CircleCrop()).into(binding.image)
}

override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
}

```

One last thing I have promised for a simpler solution to the fragment view binding.

Take AutoClearedValue.kt and copy it into your utility package. Until google will add this property delegate we will use this. This property delegate keeps track of the fragment lifecycle and upon destruction of the fragment update null value in the property

Use the by autoCleared() for your binding properties and remove the _binding and its related code.

```
//private var _binding : CharactersFragmentBinding? = null
private var binding : CharactersFragmentBinding by autoCleared()
// private val binding get() = _binding!!

private lateinit var adapter: CharactersAdapter

override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    // _binding = CharactersFragmentBinding.inflate(inflater,container,false)

    binding = CharactersFragmentBinding.inflate(inflater,container, attachToParent: false)
    return binding.root
}
```


Application Components Part 1 - BroadcastReceiver and AlarmManager

Download the Components Intro PDF:

<https://drive.google.com/file/d/1JUKSJZweYdXwtLvNGm1NUeSsw5mtHGQr/view?usp=sharing>

Download the Broadcast Receivers PDF Guide:

https://drive.google.com/file/d/1Ba_I7jxjblGIKueFyQOu3Fq3LsSDT0cN/view?usp=sharing

Download the AlarmManager PDF Guide:

<https://drive.google.com/file/d/1NuhllQqgeqHWJqJhP8s5F5nmIKbaE8g6/view?usp=sharing>

Download the Application Components Starter Project:

<https://drive.google.com/file/d/1AM06Rq1-6RJdoObCMorXBijG7ZluV0ES/view?usp=sharing>

Download the Application Components 1 final App

<https://drive.google.com/file/d/1d4dPRnzxqFPrAkXp8E5YAOBD8gmwFj7I/view?usp=sharing>

Introduction

This document's main subjects are **Broadcast component**, **AlarmManager**, **JobScheduler**, **Service component** and the **WorkManager API**. We will cover these subjects by writing an app in Kotlin that follows the **Android 8.0+** limitations. We will use various features and libraries such as view binding by delegation, Navigation component and different types of coroutines. implementing these features helps us write an app that is maintainable, reusable, and readable.

This document comes with a Kotlin project starter that contains pre-existing code and resources. In the course of the tutorials, we will implement the necessary code for making the app work, But before we dive in, we are going to go over the important parts of the existing code.

enjoy

Pre – existing code

View binding by delegation

To make the code concise, we are going to use the **Kotlin Delegated property**.

"In software engineering, the delegation pattern is an object-oriented design pattern that allows object composition to achieve the same code reuse as inheritance."

(Wikipedia)

In our context, we can use delegation Instead of inheritance to reuse the viewLifecycleOwner's onDestroy() method and null out the binding reference.

Kotlin supports this design pattern by making use of the "by" keyword (as in "provided by"). This keyword means that the get() and set() methods of a property are implemented by a different class. Essentially, we are redirecting the work that needs to be done to a different part of the code. An analogy for this concept can be an employee giving his assignment to someone else.

Let's take a look at the following code:

```
private var _binding: FragmentBluetoothBinding? = null
private val binding get() = _binding!!

override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
```

In each fragment we need to write these lines manually. Instead, we can create a new class and observe the fragment's view lifecycle. Once it gets to onDestroy() , we null out the binding. The following line of code is an example of using Kotlin Delegation property.

```
// view binding by delegation
private var binding: FragmentBluetoothBinding by autoCleared()
```

Every fragment in the application uses this feature, **and there is no need to re-write it.**

AppUtils class

For this project we have prepared a utility class that contains the following methods:

`playsound(context: Context)` – plays the system default notification ringtone.

`makeToast(context: Context, msg: String)` – shows a toast with the given message.

`notify(context: Context, title: String, msg: String, iconRes: Int? = null)` – shows a notification. If no icon was set, then it will use the `ic_favorite` icon – the default icon for our notifications.

`showSnackbar(view: View, data: String)` – method for showing a snackbar for a short amount of time. We pass a view so we can find its parent in order to display it on the screen, and a string as content.

Use these functions when needed through the AppUtils class. For example:

```
AppUtils.makeToast(context, "this is a message")
```

```
notify(context, "Harry Potter", "You're a wizard harry!", R.drawable.owl)
```

We will also add another method for creating a **notification channel**. More on that later in the AlarmManager section.

Android Notifications

A notification is a message that Android displays outside your app's UI to provide the user with reminders, communication from other people, or other timely information about events in your app. Users can tap the notification to open your app or take an action directly from the notification.

A basic notification contains an icon, title, and a small amount of content. In this small part of the tutorial, we will go over creating a notification. To get started, We need the `NotificationCompat.Builder`. We use the compatible version to support devices with lower versions. **The following code is already present in the project.**

```
fun notify(context: Context, title: String, msg: String, iconRes: Int? = null, ) {  
  
    val build = NotificationCompat.Builder(context, NOTIFICATION_CHANNEL_ID)  
        .setContentTitle(title)  
        .setContentText(msg)  
        .setPriority(NotificationCompat.PRIORITY_HIGH)  
  
    if (iconRes != null) {  
  
        build.setSmallIcon(iconRes)  
    }else{  
  
        build.setSmallIcon(R.drawable.ic_favorite)  
    }  
  
    NotificationManagerCompat.from(context)  
        .notify(NOTIFICATION_ID, build.build())  
}
```

In order to build a notification we need:

- A title.
- The message (The body of the notification).
- An icon – In the method above, if no icon was passed, than it uses the `ic_favorites` drawable.
- Priority – The importance of notification (explained underneath).

Android 8.0+ Notification Limitations:

Beginning with Android 8.1 (API level 27), apps cannot make a notification sound more than once per second. If your app posts multiple notifications in one second, they all appear as expected, but only the first notification per second makes a sound. Notice we also provide a notification channel. This is part of the Android 8.0 updates that will be covered during this course.

Message Importance Priority

Android uses the priority parameter to determine how much the notification should interrupt the user both visually and audibly. The higher the importance, the more interruptive it will be. The following are the supported levels:

Priority	Behavior	Explanation	Use Case
DEFAULT	Sound	Not interusive, Seen at the user's earliest convenience.	Traffic alerts, Taks
MIN	No sound, And now visuals	None-essential information	Weather, Ads
LOW	No sound	Less important information	new content in subsriptions
HIGH	Sound + visuals	Most importnat information	Text messages, phone calls, alarms

Custom Notification View

Android Provides different templates with different actions for notifications, that allows us to achieve complex interactions with little effort. For example, Use the [MessagingStyle](#) class to display text messages.

However, if the system templates don't meet your needs, you can provide you own layout. After you create the layout, instead of calling methods to set the title and the body, call the `setContent()` method. This method receives our package name and **RemoteView**.

```
// Get the layouts to use in the custom notification
val notificationLayout = RemoteViews(packageName, R.layout.notification_small)
```

```
val build = NotificationCompat.Builder(context, NOTIFICATION_CHANNEL_ID)
    .setContent(notificationLayout)
    .setPriority(NotificationCompat.PRIORITY_HIGH)
```

Navigation component

In 2018, Google introduced a new architecture component as part of the Android Jetpack library. By using this component, we can manage the navigation of the fragments back and forth within a single XML file, making it easy to maintain the app navigation.

The navigation component uses a navigation resource file to represent the screens flow as a graph. The graph contains all the app's destinations (fragments and activities) along with their animations and pop behavior. Instead of using a fragmentManager with a fragmentTransaction, we will handle the transactions in the navigation component, and perform a navigation action when called.

The dependencies, and the navigation graph is already included in the project, no need to re- write it

Broadcast receivers

Broadcast receivers act like car antennas. Just like antennas can be configured to receive a specific frequency and listen for a radio station, a broadcast receiver can be configured to receive a specific intent action.

A broadcast is an event that the Android system can send when events occur. By registering to these events, we can activate our own methods and components. For example, the Android OS triggers an event when the screen turns on and off, and by setting up a broadcast receiver, we can play a nice ringtone every time the screen's status changes.

We can also make a custom broadcast and define it to be exclusive to our app so components outside of our app won't be able to receive it. That way we can create this private "message board" for our app and deliver messages from one component to another. **In general, broadcasts are messaging components used for communicating across apps when events of interest occur.**

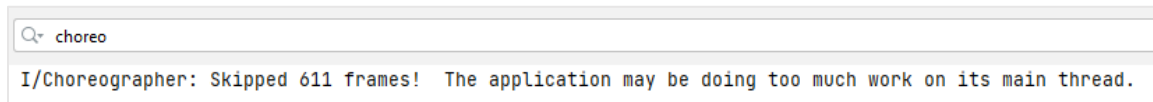
We can register for these kinds of events by setting up a broadcast receiver and declare it in the manifest file, or register it dynamically in our code.

Note: The broadcast receiver works on the main thread. Broadcast receivers are allowed to run for up to 10 seconds before the system will consider them as non-responsive, and decide to terminate them. The main thread itself has a limitation of 5 seconds to be unresponsive, so if we opt out into a background thread, the 10 second rule will apply.

Bonus

Try in any app, to activate the `Thread.sleep()` function for 5-10 seconds. Then press the different views in the UI. The app will not respond and after a while, it will simply crash with a message that it's not responding.

When there is too much work on the main thread, we will see a message in Logcat from the Choreographer:



The Choreographer coordinates the timing of the UI – animations, input and drawings. It does that by receiving timing pulses from the display system and then scheduling work to be done for the next frame.

In the picture above, the choreographer outputs a warning that 611 frames were skipped. Most devices have a refresh rate of 60 frames per second. So Basically the app skipped about 10 seconds. That's terrible! Be aware of where you work.

Implementation:

Before we start writing the Broadcast receivers, let's prompt the user to enable Bluetooth. We will use an `ActivityResultLauncher`, through a `suspendCoroutine`. By using this launcher, we can fire an `ActivityResultContract`, which is an abstraction for the `onActivityResult()` method, that lets us handle the result in a clean and reusable way. [ActivityResultContracts](#) is a class that contains a collection of contracts.

A contract can be thought of as an agreement between the class and the rest of the system: how the user should interact with it, and what it promises back.

Moreover, we can even decouple the code responsible for managing the `ActivityResult`. Meaning, our fragments and activities will be cleaner. By providing a reference to the [ActivityResultRegistry](#), we can register our callbacks from any class, instead of being bound to register from a fragment or an activity. To register, use the `register()` method. This method is actually called by `registerForActivityResult()` method.

We make sure to unregister the launcher when no longer needed.

The project contains a `BluetoothHandler` class that registers for an `ActivityResultContracts` that lets the users enable (or not) the Bluetooth component in their phones. This method uses a `suspendCoroutine` to achieve that.

```
suspend fun requestBluetoothActivation(): Boolean {  
  
    return suspendCoroutine { continuation ->  
        val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)  
        launcher = resultRegistry.register(  
            BLUETOOTH_REQUEST,  
            ActivityResultContracts.StartActivityForResult()  
        ) { result ->  
            continuation.resume(result.resultCode == Activity.RESULT_OK)  
        }  
  
        launcher!!.launch(enableBtIntent)  
    }  
}  
  
fun unregister() {  
    launcher?.unregister()  
}  
  
BluetoothHandler.kt
```

Through the `resultRegistry` we use the `register()` method. We provide the `register()` method with a key (`BLUETOOTH_REQUEST`) to identify the Bluetooth enable request call. We also pass a contract (`StartActivityForResult()`), and provide a callback to handle the results. We also need to unregister the launcher when no longer needed:

```
override fun onDestroyView() {  
    super.onDestroyView()  
    permissionRequests.unregister()  
}  
  
BluetoothFragment.kt
```

suspendCoroutine

This function was designed for conversion of callbacks into suspending functions, but is rarely used since it exposes a **Continuation** object (the trick behind the “magic”). The way functions suspend is by storing the state of a function within the continuation object. The compiler creates a continuation type containing the fields of the function, and changes the function signatures to accept a continuation object. This way we can save the function’s state and allow it to resume at a later time.

This function does exactly what its name says. It suspends the coroutine that it was called from (must be called from a CoroutineScope or another suspend function), and provides a way to resume that function. So once we have the result from the `ActivityResultCallback`, we can resume it by calling the Continuation’s object `resume()` method. The value we pass inside the `resume()` will be returned back.

That’s just sounds like a callback with extra steps... but when we need to chain asynchronous functions that use callbacks, we might find ourselves in a **callback hell**, where each function is called inside the callback of the previous one. So `suspendCoroutine` function enables us to connect callbacks and coroutines together, making our code seem synchronous and easy to read.

Suppose we have 3 functions: `wakeup()`, `doWork()` and `goToSleep()`. Each of these functions do some kind of asynchronous work and provide a callback to their result. But they **do not support** suspend (plain Java / Android functions). Our goal is to make them work sequentially: First we wake up, then we do some work, and only then we go to sleep. We could start the `doWork()` method in the callback of the `wakUp()` method and the `goToSleep()` method in the `doWork()` method, but that is as tedious as it sounds, and would be hard to read and maintain.

The solution is to wrap each of these function in a `suspendCoroutine`, and call `resume()` in their callback. We can even pass a value inside `resume()`. This will let us write code in a clean way. No interfaces and no callbacks needed. Everything is under the hood and we have a nice sequential code.

```
suspend fun life(){
    repeat( times: 365){ it: Int
        wakeup()
        doWork()
        goToSleep()
    }
}
```

Let's begin the broadcast subject with a few Bluetooth related broadcasts and see how we can register for these events. The events we are interested in are:

- ACTION_STATE_CHANGED
- ACL_CONNECTED
- ACL_DISCONNECTED

For now, we start with something simple, a ringtone when any of these broadcasts are received.

The BroadcastReceiver

Create a new class inside the `data.services.utils.receivers` package and call it `BluetoothStateReceiver`. In order to make this class a Broadcast receiver, we need to inherit from the `BroadcastReceiver` class, and override the `onReceive()` method:

```
class BluetoothStateReceiver: BroadcastReceiver() {  
    override fun onReceive(context: Context?, intent: Intent?) {  
        TODO(reason: "Not yet implemented")  
    }  
}
```

Note: when generating the `onReceive()` method, the parameters names will be "p0" and "p1". This is related to the compiled SDK version in the apps gradle file. If we change the version to 30 and below, the parameters names will be normal. But for now, You may change their names for readability purpose, but not their type.

We need to differentiate between the different actions our receiver receives, so let's write it first. We will write it inside a `when` block:

```
override fun onReceive(context: Context?, intent: Intent?) {  
    when (intent?.action) {  
        BluetoothAdapter.ACTION_STATE_CHANGED -> {  
        }  
        BluetoothDevice.ACTION_ACL_CONNECTED -> {  
        }  
        BluetoothDevice.ACTION_ACL_DISCONNECTED -> {  
        }  
    }  
}
```

let's begin with the `ACL_CONNECTED` and `ACL_DISCONNECTED` first. These two differentiate between a Bluetooth device connection and disconnection. For them, we're just going to play a sound.

`onReceive` should look like this now:

```

override fun onReceive(context: Context?, intent: Intent?) {

    when (intent?.action) {
        BluetoothAdapter.ACTION_STATE_CHANGED -> {

        }
        BluetoothDevice.ACTION_ACL_CONNECTED -> {
            AppUtils.playSound(context)
        }
        BluetoothDevice.ACTION_ACL_DISCONNECTED -> {
            AppUtils.playSound(context)
        }
    }
}

```

Now we need to declare our receiver in the manifest file:

```

<receiver
    android:name=".data.utils.receivers.BluetoothStateReceiver"
    android:exported="true">
    <intent-filter>
        <action android:name="android.bluetooth.device.action.ACL_CONNECTED" />
        <action android:name="android.bluetooth.device.action.ACL_DISCONNECTED"/>
    </intent-filter>
</receiver>

```

`android:name` – the name of our receiver.

`android:exported` - If set to true, our receiver will be able to receive messages from outside the app.

`<intent-filter>` - this filter declares which intents should be received by the receiver.

`<action>` - the action we would like to register

You can run the app now, and connect to a Bluetooth device, or disconnect from it. You should hear a sound when connecting and disconnecting from a Bluetooth device.

Now let's handle the `ACTION_STATE_CHANGED`. Write a `handleStateChange()` method that will differentiate between `BluetoothAdapter.STATE_OFF`, and `BluetoothAdapter.STATE_ON`. This function will get the context and the intent:

```
private fun handleStateChange(context: Context?, intent: Intent?) {

    val state = intent?.getIntExtra(BluetoothAdapter.EXTRA_STATE, BluetoothAdapter.ERROR);
    when (state) {
        BluetoothAdapter.STATE_OFF -> {

        }
        BluetoothAdapter.STATE_ON -> {

        }
    }
}
```

Notice we get the state from an `EXTRA` field inside the intent.

Now, inside the appropriate place, call the `playsound()`.

Now we can easily handle the Bluetooth state. Call the `handleStateChange()` method in the appropriate part of the `onReceive()` method. (hint: it's where it says state changed...)

`onReceive` should look like this now:

```
override fun onReceive(context: Context?, intent: Intent?) {

    when (intent?.action) {
        BluetoothAdapter.ACTION_STATE_CHANGED -> {
            handleStateChange(context, intent)
        }
        BluetoothDevice.ACTION_ACL_CONNECTED -> {
            AppUtils.playSound(context)
        }
        BluetoothDevice.ACTION_ACL_DISCONNECTED -> {
            AppUtils.playSound(context)
        }
    }
}
```

Don't forget to add the action in our declared receiver inside the manifest:

```
<receiver
  android:name=".data.utils.receivers.BluetoothStateReceiver"
  android:exported="true">
  <intent-filter>
    <action android:name="android.bluetooth.adapter.action.STATE_CHANGED"/>
    <action android:name="android.bluetooth.device.action.ACL_CONNECTED" />
    <action android:name="android.bluetooth.device.action.ACL_DISCONNECTED"/>
  </intent-filter>
</receiver>
```

You can run the app now, and switch the Bluetooth on and off. If your device is running Android version 11 and above, you will notice that the Bluetooth state change is not received.

Like many other broadcasts, this broadcast will not be received if we register for it in the manifest. In order to receive them, we need to register **Dynamically**. These limitations began in Android 8 and got more strict in the next versions.

Android 8.0+ implicit broadcast limitations:

Beginning with Android 8.0, the Android OS enforces additional limitations on manifest-declared receivers. If our app targets Android 8.0 and above, **we can't use the manifest to declare receivers for most implicit broadcasts**. However, apps can continue to register receivers for the broadcasts. There are some implicit broadcasts that are exempted from this limitation, check the following link:

<https://developer.android.com/guide/components/broadcast-exceptions>

If an app registers to receive broadcasts, the app's receiver uses resources and memory every time the broadcast is received. If we declare that receiver in the manifest, then the **app doesn't even have to be alive for the receiver to get triggered**.

Let's assume there are more apps in our device that are registered to this broadcast. That means that whenever this broadcast is sent, all of the receivers inside the apps (declared in manifest) will get triggered and consume a lot of resources. This is obviously a problem. The device's battery will be depleted relatively fast, and the user experience overall will be impacted badly.

For example, the ACTION_STATE_CHANGED event. We cannot register for it in the manifest, but we can do it in Dynamic receivers. **Dynamic receivers** are registered using an application context or an activity context. If we use the application context to

register the receiver, our app receives broadcasts as long as the app is alive. That way, we can register for any broadcast. If we use an activity context to register the receiver, the app receives broadcasts until the activity is destroyed.

Let's get back to our code, and register for the `ACTION_STATE_CHANGED` dynamically. Inside the `BluetoothFragment` class, create an instance of the `bluetoothStateReceiver`, and a `handleBluetoothConnection()` method. This method should instantiate the instance of our receiver, and register it with an `IntentFilter`. The constant we plant in the `IntentFilter` will enable us to register for the `ACTION_STATE_CHANGED` broadcast.

```
private lateinit var bluetoothConnectionReceiver: BluetoothStateReceiver

private fun handleBluetoothConnection() {
    val filter = IntentFilter(BluetoothAdapter.ACTION_STATE_CHANGED)
    bluetoothConnectionReceiver = BluetoothStateReceiver()
    requireActivity().registerReceiver(bluetoothConnectionReceiver, filter)
}
```

Call this method from the `onViewCreated` in the `BluetoothFragment` class. Also, We need to unregister the dynamic receiver in order to avoid memory leaks and unexpected behavior

```
override fun onDestroyView() {
    super.onDestroyView()
    permissionRequests.unregister()
    requireActivity().unregisterReceiver(bluetoothConnectionReceiver)
}
```

You may delete the `STATE_CHANGED` action from the receiver in the manifest, and run the app. test the `ACTION_STATE_CHANGED` broadcast by switching the Bluetooth on and off, you should hear a sound. If you quit the app and try to test it, nothing will happen because the receiver is registered via our activity's context, and the activity is no longer alive.

Let's handle another broadcast - the `ACTION_AIRPLANE_MODE_CHANGED` broadcast.

Create a new class and call it `AirplaneModeReceiver`. As before, we need to inherit from the `BroadcastReceiver` class, and override the `onReceive()` method:

```
class AirplaneModeReceiver : BroadcastReceiver() {  
  
    override fun onReceive(context: Context?, intent: Intent?) {  
        TODO(reason: "Not yet implemented")  
    }  
}
```

Like in the `BluetoothStateReceiver` class, we are going to play a ringtone when we receive a broadcast. Get the action from the intent, and check that it matches the relevant broadcast constant. If it does, call the `playRingtone()` method:

```
class AirplaneModeReceiver : BroadcastReceiver() {  
  
    override fun onReceive(context: Context?, intent: Intent?) {  
        val airplaneAction = Intent.ACTION_AIRPLANE_MODE_CHANGED  
        if (intent?.action.equals(airplaneAction)){  
            AppUtils.playSound(context)  
        }  
    }  
}
```

In order to show the Android 8.0 broadcast limitations, try to register the `airplaneModeReceiver` in the manifest, and run the app:

```
<receiver android:name=".data.utils.receivers.AirplaneModeReceiver"  
    android:exported="true">  
    <intent-filter>  
        <action android:name="android.intent.action.AIRPLANE_MODE"/>  
    </intent-filter>  
</receiver>
```

Notice that it doesn't work. Now that we've seen how we should register this kind of broadcast, let's register and unregister it **dynamically**. You may now delete the receiver you just declared in the manifest.

Let's take a look at the `BluetoothFragment` fragment. Create an instance of the `airplaneModeReceiver`, and a `handleAirplaneMode()` method. This method should instantiate the instance of our receiver, and register it with an `IntentFilter`. The constant we plant in the `IntentFilter` will enable us to register for the `AIRPLANE_MODE_CHANGED` broadcast.

```
// Dynamic BroadcastReceiver
private lateinit var airplaneModeReceiver: BroadcastReceiver

private fun handleAirplaneMode() {
    val intentFilter = IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED)
    airplaneModeReceiver = AirplaneModeReceiver()
    requireActivity().registerReceiver(airplaneModeReceiver, intentFilter)
}
```

We must unregister our receiver to **avoid memory leaks and unexpected app behavior**. Unregister receivers when the app no longer needs them, or before the activity is destroyed. In the `onDestroyView()` method in the same fragment, write the following:

```
override fun onDestroyView() {
    super.onDestroyView()
    permissionRequests.unregister()
    requireActivity().unregisterReceiver(blueToothConnectionReceiver)
    requireActivity().unregisterReceiver(airplaneModeReceiver)
}
```

After writing this method, call it from the `onViewCreated()` function of the `BluetoothFragment` fragment.

Now, as long as our activity is alive, the receiver will get the broadcast and activate a ringtone sound. Run the app and turn airplane mode on and off. The phone's default ringtone should be played. Try to exit the activity (we only have one so just exit the app itself), and notice that it stopped working. This is because we registered this receiver with the activity's context, and it's not alive anymore.

Custom broadcasts and LocalBroadcastReceivers

Custom broadcasts are messages that our app can send to the rest of the system, and any app that has the action string can register for it. We use a custom broadcast when we want our app to do something without explicitly launching an activity. For example, we can send a broadcast when we want to notify other apps in the device that new data has been received, so they would do something accordingly.

LocalBroadcast receivers are receivers for our app only. Unlike regular receivers, other apps cannot send broadcast to this receiver. We use the **LocalBroadcastManager** to set them up. This way, broadcasted data will only be visible to our app's components, and it's considered more efficient because there's no IPC (inter-process communication), so less overhead of the broadcast.

For now, we will show a simple snackbar with the data we received in the `onReceive()` method. The next part will show you how to write a local broadcast receiver (it's just like a normal receiver), and how to register it with a custom intent filter (and unregister as well). We are going to send a custom broadcast to ourselves, and once we receive the data in the receiver, we will pass it to our fragment, and show it in a snackbar. We use two constants for this part: one for configuring the custom action string, and the other as a key for our data. These constants are present in the `BluetoothFragment.kt` class, and we will use them when needed.

Create a new class in the `data.utils.receivers` package, and call it `LocalBroadcastReceiver`. Just like the previous receivers, we need to inherit from `BroadcastReceiver` class, and override the `onReceive()` method:

```
class LocalBroadcastReceiver: BroadcastReceiver() {  
  
    override fun onReceive(context: Context?, intent: Intent?) {  
        TODO(reason: "Not yet implemented")  
    }  
}
```

In the `onReceive()` method, we need to check that the broadcast we receive matches the custom action string that we will provide when we set the intent filter from the `BluetoothFragment`.

```

override fun onReceive(context: Context?, intent: Intent?) {
    if (intent?.action==BluetoothFragment.CUSTOM_ACTION){
        val data = intent.getStringExtra(BluetoothFragment.DATA_EXTRA)
        data?.let { it: String
            }
        }
    }
}

```

The `CUSTOM_ACTION` string is simply a string that consists of our package's names. The action string that we need to provide must be unique so there won't be any conflicts with other receivers. we can use our package name for that. The `DATA_EXTRA` is also a string. Both of them are inside a companion object in the `BluetoothFragment.kt` class:

```

companion object {
    const val CUSTOM_ACTION = "com.eran.applicationcomponents.action.custom_broadcast"
    const val DATA_EXTRA = "custom_broadcast_action"
}

```

Next, lets create an interface with one method in it. We will use this interface to send the data to the BluetoothFragment. For that, we also have to pass a callback parameter in our `LocalBroadcastReceiver` class. Change the `LocalBroadcastReceiver` class as follow:

```

class LocalBroadcastReceiver(private val callback: LocalReceiverCallback): BroadcastReceiver() {
    interface LocalReceiverCallback {
        fun onDataReceived(data: String)
    }

    override fun onReceive(context: Context?, intent: Intent?) {
        if (intent?.action == BluetoothFragment.CUSTOM_ACTION) {
            val data = intent.getStringExtra(BluetoothFragment.DATA_EXTRA)
            data?.let { it: String
                callback.onDataReceived(it)
            }
        }
    }
}

```

Now Let's work on the fragment side. Create an instance of the `LocalBroadcastReceiver` inside the `BluetoothFragment.kt` class:

```

// local broadcastReceiver
private lateinit var localReceiver: LocalBroadcastReceiver

```

We will instantiate and register it in a new method: `handleLocalBroadcast()`. In order to pass the data to the fragment, we need to pass a **callback object** in the constructor of the `LocalBroadcastReceiver`, and implement the `onDataReceived()` method we defined in the interface. Inside the callback, call the `AppUtils.showSnackbar()` method, and pass the data that we received in the callback:

```
private fun handleLocalBroadcast() {
    localReceiver =
        LocalBroadcastReceiver(object : LocalBroadcastReceiver.LocalReceiverCallback {
            override fun onDataReceived(data: String) {
                AppUtils.showSnackbar(binding.nextPage, data)
            }
        })
}
```

Now we need to register and unregister our receiver. We will register the receiver in the `handleLocalBroadcast()` method, using an intent filter with our custom action string, and unregister it in the `onDestroyView()` method of the `BluetoothFragment`. Once done, call the `handleLocalBroadcast()` from the `onViewCreated()` method

```
private fun handleLocalBroadcast(){
    localReceiver=
        LocalBroadcastReceiver(object: LocalBroadcastReceiver.LocalReceiverCallback{
            override fun onDataReceived(data: String) {
                AppUtils.showSnackbar(binding.nextPage, data)
            }
        })
    val filter = IntentFilter(CUSTOM_ACTION)
    LocalBroadcastManager.getInstance(requireActivity()).registerReceiver(localReceiver, filter)
}
```

```
override fun onDestroyView() {
    super.onDestroyView()
    permissionRequests.unregister()
    requireActivity().unregisterReceiver(bluetoothConnectionReceiver)
    requireActivity().unregisterReceiver(airplaneModeReceiver)
    LocalBroadcastManager.getInstance(requireActivity()).unregisterReceiver(localReceiver)
}
```

Before we continue any further, lets use some Kotlin goodness to make our callback code more concise:

Functional interface (SAM)

SAM (Single Abstract Method) were introduced in Kotlin version 1.4. this kind of interface has a single abstract method in it, and is also called functional interfaces. By using this type of interface with the help of **SAM conversions**, we reduce boilerplate code, meaning we reduce code that we repeat a lot, while keeping it very much readable. SAM conversions uses lambda expressions which is a block of code that can be passed to another function. These type of functions that can accept functions as parameters are called **Higher Order Functions**.

To declare a functional interface, we use the `fun` keyword before the `interface`. **the interface must contain only one abstract method**. Let's change our code to use the SAM interface. in the `LocalBroadcastReceiver.kt` class, add the keyword `fun` before the `interface` keyword.

```
fun interface LocalReceiverCallback {
    fun onDataReceived(data: String)
}
```

Now in the `BluetoothFragment` where we defined the callback, notice the compiler warns us to convert the callback to a lambda expression:

```
private fun handleLocalBroadcast() {
    localReceiver =
        LocalBroadcastReceiver(object : LocalBroadcastReceiver.LocalReceiverCallback {
            override fun onDataReceived(
                AppUtils.showSnackBar(binding)
            )
        })
    val filter = IntentFilter(BluetoothF
        LocalBroadcastManager.getInstance(re
    }
}
```

Convert to lambda
Convert to lambda Alt+Shift+Enter More actions... Alt+Enter
com.eran.applicationcomponents.data.utils.receivers LocalBroadcastReceiver.kt
public final class LocalBroadcastReceiver : BroadcastReceiver
ApplicationComponents.app

Click on the **Convert to lambda** text. We now have a much more concise and easy to read code. Lambda expressions are awesome!

```
private fun handleLocalBroadcast() {
    localReceiver =
        LocalBroadcastReceiver { data ->
            AppUtils.showSnackBar(binding.nextPage, data)
        }
    val filter = IntentFilter(BluetoothFragment.CUSTOM_ACTION)
    LocalBroadcastManager.getInstance(requireActivity()).registerReceiver(localReceiver, filter)
}
```

Back to the code! Now, Let's create a `sendCustomBroadcast()` method to send our own broadcast for the local receiver. Inside the intent that we will use to send the broadcast, We will use "DATA_EXTRA" (the same one as before) as the key for our data in the intent's `putExtra()` method, and "CUSTOM_ACTION" (already present in the class) as the key for the intent's action. Once the intent is set, we will use the `LocalBroadcastManager` to send the broadcast.

```
private fun sendCustomBroadcast() {
    val intent = Intent()
    intent.apply { this: Intent
        action = CUSTOM_ACTION
        putExtra(DATA_EXTRA, value: "this is my custom broadcast to my local receiver!")
    }

    // send it
    LocalBroadcastManager.getInstance(requireContext()).sendBroadcast(intent)
}
```

Also, let's add a button with a listener that when clicked, it will send our custom broadcast. The xml code is already in the project, just add the listener.

In the `onViewCtread()` method of `BluetoothFragment`:

```
binding.sendBroadcast.setOnClickListener {
    sendCustomBroadcast()
}
```

Run the app. Now, when we click on the send broadcast button, a snackbar will appear with the value that we put inside the intent that we sent.

Sending a broadcast with permissions

we can restrict broadcasts to apps that hold specific permissions. We can enforce these restrictions on either the sender or the receiver of a broadcast. The permissions are specified in the optional parameters of the `sendBroadcast()` method or the `sendOrderedBroadcast()` method. **That way, only receivers who have been granted the permissions can receive the broadcast.**

The following snippet (**is not in the project**) shows how to send a broadcast with Bluetooth access permission:

```
val customIntent = Intent()
customIntent.apply {

    // Set a unique action string with our package name as a prefix.
    action = "com.myapp.broadcast.MY_NOTIFICATION"

    // add your data to the intent
    putExtra("data", "bluetooth is pretty awesome too!!")
}

// send the intent with a bluetooth access permission
requireActivity()
    .sendBroadcast(customIntent, Manifest.permission.BLUETOOTH)
```

Best practices

- If we need to do long running tasks, we should be aware that creating new threads and background services in the `onReceive()` may be **killed by the system** once the method returns. To perform long running tasks, we will use the JobScheduler (shown on JobScheduler section)
- Namespaces for broadcast actions are system - global. Make sure your action names are unique to prevent conflicts with other apps.
- When registering a receiver, any app can send malicious broadcasts to our app's receiver. Use LocalBroadcast when possible and specify a permission when registering.
- Do not broadcast sensitive data using an implicit intent. The information can be read by any app that registers to receive the broadcast.
- Use context registration instead of manifest declaration when possible. If many apps have registered to the same broadcast, it can cause the system to launch a lot of apps, making a bad impact on performance and user experience.

Alarm Manager

In this section, we will talk about the [AlarmManager](#). The AlarmManager goes hand in hand with the broadcast receivers since by using the **AlarmManager**, we set specific code to run in the future through a receiver. Once the alarm goes off, a broadcast will be sent to the broadcast receiver, and the code will be executed. **The AlarmManager allows an application to perform some functions even after the application process or all of its Android components have been cleaned up by the system.** This is because the AlarmManager provides access to system – level alarm services

For now, we only use the AlarmManager to **display a notification**, and demonstrate how it can be triggered even after we quit the app. The app already contains a class named `AlarmManagerReceiver` and we will add the necessary code to make it work.

But before we can start sending notifications, we must register the notification to a **Notification Channel**

Android 8.0+ Notification Channels:

Notification Channels is a feature introduced in Android 8. Each notification must be assigned to a channel. These channels allow the users to have more control over the notifications. We can set the notification settings to whichever we like, but once the channel is created, the control over the notification goes to the user. As mentioned in the AppUtils section, we are going to write a `createNotificationChannel()` method, and activate it in our `MainActivity.kt` class, so when the app starts it will create the channel, and we can make notifications according to the Android 8.0 requirements. Inside the AppUtils' companion object, add the following method

```
fun createNotificationChannel(context: Context) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val channel = NotificationChannel(
            NOTIFICATION_CHANNEL_ID,
            NOTIFICATION_CHANNEL_NAME,
            NotificationManager.IMPORTANCE_HIGH
        )

        NotificationManagerCompat.from(context).createNotificationChannel(channel)
    }
}
```

Notice in the `createNotificationChannel()` we check the `SDK_INT` to be Android Version 0 and above. For a full list of changes done in Android 8.0 visit:

https://developer.android.com/reference/android/os/Build.VERSION_CODES.html#0

Now, let's create the `AlarmManagerReceiver` class. As usual, inherit from the `BroadcastReceiver` class, and override the `onReceive()` method. In the `onReceive()` method, call the `AppUtils.notify()` method. You may use any title and text you want, or use the preexisting ones:

```
class AlarmManagerReceiver : BroadcastReceiver() {  
    override fun onReceive(context: Context?, intent: Intent?) {  
        AppUtils.notify(  
            context!!,  
            context.getString(R.string.notif_title),  
            context.getString(R.string.notif_text)  
        )  
    }  
}
```

Next, we need to declare the receiver in the manifest because it is a component that is activated by an intent.

```
<receiver android:name=".data.utils.receivers.AlarmManagerReceiver" />
```

Next, let's work on the AlarmFragment. The xml layout file is already present, no need to create one. This layout contains one button that we will use to set the alarm

First, declare a PendingIntent variable:

```
private lateinit var setAlarmIntent: PendingIntent
```

In `onViewCreated()`:

```
val intent = Intent(requireActivity(), AlarmManagerReceiver::class.java)
setAlarmIntent = PendingIntent.getBroadcast(
    requireActivity(),
    ALARM_REQUEST_CODE,
    intent,
    PendingIntent.FLAG_UPDATE_CURRENT
)
```

Create a new method: `handleAlarm()`:

```
private fun handleAlarm() {
    val alarmManager =
        requireActivity().getSystemService(Context.ALARM_SERVICE) as AlarmManager

    val timeOfAlarm = SystemClock.elapsedRealtime() + 10000L // alert in 10 seconds
    alarmManager.setExact(AlarmManager.ELAPSED_REALTIME_WAKEUP, timeOfAlarm, setAlarmIntent)
}
```

`SetExact()` – This sets the alarm to fire right after the given milli seconds. Providing a time in the past will trigger the alarm immediately. If there is already an alarm scheduled for this intent, then the new alarm will take its place. The alarm will be triggered as nearly as possible to the requested alarm time.

Note I: Apps that target device SDK version 31 and higher must be granted the SCHEDULE_EXACT_ALARM permission (already added in the manifest):

```
<uses-permission
android:name="android.permission.SCHEDULE_EXACT_ALARM"/>
```

Note II: there is also the `set()` method which is inexact. Up until then, this method delivered the alarm in an exact manner. Beginning from SDK 19 The system may defer and trigger it at a later time in order to save battery and optimize background work. For now we will use `setExact()`.

`SystemClock.elapsedRealtime()` – return milliseconds since boot, including sleep time.

`ELAPSED_REALTIME_WAKEUP` – This alarm uses the elapsed time since the device boot (including sleep time).

There are more constants that will make the alarm act differently:

- **RTC_WAKEUP** – this constant will trigger the alarm even if the device is asleep. The alarm time is in milliseconds, and uses wall clock time (UTC), meaning it uses the current time.
- **RTC** – Also in wall clock time. This alarm does not wake the device, and will be delivered only in the next time the device wakes up.
- **ELAPSED_REALTIME** – Same as `ELAPSED_REALTIME_WAKEUP` but does not wake the device up.

Pay attention to which type of clock time is needed. If declared type `ELAPSED_REALTIME` (wake up or not), then we should use the `SystemClock.elapsedRealtime()`. If declared `RTC` (wake up or not), then we should use the `System.currentTimeMillis()`.

You can read more about the different alarm type here:

https://developer.android.com/reference/android/app/AlarmManager.html#RTC_WAKEUP

Note: In Android 12 (and upcoming versions), a special permission is needed in order to set **exact alarms**. Exact alarms are triggered at a precise moment in the future.

```
<uses-permission
android:name="android.permission.SCHEDULE_EXACT_ALARM"/>
```

The user can revoke this permission. so in order to get permissions back, we need to check if we still have the permission by using the `canScheduleExactAlarms()` method and fire an intent that includes the [ACTION_REQUEST_SCHEDULE_EXACT_ALARM](#) intent action.

Back to our code. Call the `handleAlarm()` method that you wrote inside the `onClickListener` of the alarm button, in the `onClickListener` of the alarm button (`AlarmFragment.kt`):

```
binding.alarm.apply { this: MaterialButton
    setOnClickListener { it: View!
        handleAlarm()
        showDisableBtn()
        it.startAnimation(animation)
    }
}
```

There's also a nice animation that will play when we click the button.

Cancel Alarms

To cancel an alarm, use the `cancel()` method. We need to provide the same `PendingIntent` when we first set the alarm. Create a `disableAlarm()` method, with the same `pendingIntent` when we set the alarm:

```
private fun disableAlarm(){
    val alarmManager =
        requireActivity().getSystemService(Context.ALARM_SERVICE) as AlarmManager
    alarmManager.cancel(setAlarmIntent)
}
```

Call this function from the `disableAlarm` `onClickListener`

```
binding.disableAlarmBtn.setOnClickListener { it: View!
    disableAlarm()
    hideDisableBtn()
}
```

There are preexisting `hideDisableBtn()` and `showDisableBtn()` methods. The first one is already in the code. Let's put the latter in the `handleAlarm()` method:

Run the app. Enter the **AlarmFragment**, and click the alarm button. After 10 seconds, our notification will pop up. As mentioned in the beginning of this section, the app does not have to be alive to trigger the receiver, so we can click the button, and exit the app- the receiver will still work. Try to disable the alarm as well and see that the alarm will not trigger.

There are different ways to set an alarm. We use the simplest one to set a one-time alarm. You can also set an alarm to go off in pre-determined time intervals – inexact alarms.

InexactRepeating()

This type of alarm, as the name suggests, will not be delivered exactly when we set it up to run, but will be delivered at a time when the system thinks it's most efficient for the device's battery. In Android 12 and higher, the alarm is delivered within one hour from the trigger time. As mentioned before, the `set()` method is inexact.

Let's Change the `handleAlarm()` method to use the `InexactRepeating()` method:

```
private fun handleAlarm() {
    val alarmManager =
        requireActivity().getSystemService(Context.ALARM_SERVICE) as AlarmManager

    alarmManager.setInexactRepeating(
        AlarmManager.ELAPSED_REALTIME_WAKEUP,
        triggerAtMillis: SystemClock.elapsedRealtime() + 10000L,
        AlarmManager.INTERVAL_FIFTEEN_MINUTES,
        setAlarmIntent
    )
    showDisableBtn()

    val timeOfAlarm = SystemClock.elapsedRealtime() + 10000L // alert in 10 seconds
    alarmManager.setExact(AlarmManager.ELAPSED_REALTIME_WAKEUP, timeOfAlarm, setAlarmIntent)
}
```

This alarm will first fire when we set it, and then repeat itself every 15 minutes in an inexact manner. Run the app and test it for yourself. The app doesn't need to be alive for the alarm to work, and it will even wake the device up.

The AlarmManager is great for offsetting future tasks and even **preserves battery life** when using inexact scheduling. The Android system groups the alarms together from multiple applications, thus avoiding frequent device wake and networking. **But what if our tasks are not based on time**, but rather on factors such as whether the device is charging or not, or network access. the AlarmManger won't help us here.

You may disable the inexact alarm code and bring back the previous one for the next sections. It will be easier to test things out.

Application Components 2 - Services, JobScheduler and WorkManager

Download the Service PDF Guide here:

https://drive.google.com/file/d/1rOhuIALjGC_EO9cUQbT8rZL-NFThC0L9/view?usp=sharing

Download the JobScheduler PDF Guide here:

<https://drive.google.com/file/d/1KejxmshiS-1top6OZsOXFzcYWu21LpeG/view?usp=sharing>

Download the WorkManager PDF Guide from here:

<https://drive.google.com/file/d/1t3Hcat7eT-nNgkWAuFq0zOBnFSdCdq9E/view?usp=sharing>

Download the Components 2 Starter project:

<https://drive.google.com/file/d/1oOOhp5xQR1fwmRKHgI7O1WVbXlMunIAU/view?usp=sharing>

Download the Components 2 Final App:

<https://drive.google.com/file/d/1wjTkr3rWEGzKKp7ol4zdSfwgpg5PIN3F/view?usp=sharing>

Services

Introduction

A service is an application component that can perform long – running operations without a UI. A lot of explanations on the web about the Android service (including developer.android.com) begin with a statement that it is a component which runs in the background. That's misleading because one might think that the code inside the service automatically runs in a background thread.

So let this be clear: **A service runs in the main thread of the application. In order to actually work in a background thread inside a service, we need to create one by ourselves, either by using Thread(), or using various Kotlin coroutine methods.**

Now that we got that out of the way, let's explore the subject.

Note: A lot of functionality of the service may be implemented with the relatively new WorkManager API. So if you read something in this section and think that you might be able to implement the same functionality in a cleaner fashion with a WorkManager – you probably can.

JobScheduler

As mentioned before, we can activate our functions at a certain time, and set it to repeat itself even outside of the app's lifetime. But a lot of tasks that we might need to do in the future won't be time dependent, but rather based on user interaction and system states.

For example, an app might need to update itself, but it needs to be done when there's internet connection (obviously). Another example is if an app would like to activate a heavy service that consumes a lot of resource and battery. In this case, we might want to **constraint the trigger** of the service and activate only when the phone is charging, or simply in idle mode. AlarmManager doesn't offer these kinds of constraints, so if you want to write that code for yourself, Goodluck!

Enter **JobScheduler**. The JobScheduler works on devices with SDK version 21 and above. The JobScheduler is meant exactly for the kind of job the AlarmManager can't do. By setting a "contract" for the job, we can condition our code to fire when the contract's conditions are met. An analogy for this would be giving employees an assignment, but telling them to do it once they are not doing anything else.

Let's get familiar with some basic components of the JobScheduler.

JobInfo – This class contains a set of constraints and conditions that we want our scheduler to work under. This is the contract we have with the JobScheduler. through this class, we can get an instance of a **Builder** to define our constraints for the job. After we finish building, we pass the created JobInfo to the JobScheduler.

JobService – **This service executes each job on the main thread**, so if the task is a long or heavy one, we need to make sure to do it on a different thread. The tasks we need to run will be implemented inside this service. We need to inherit from this class and implement the `onStartJob()` and `onStopJob()` methods.

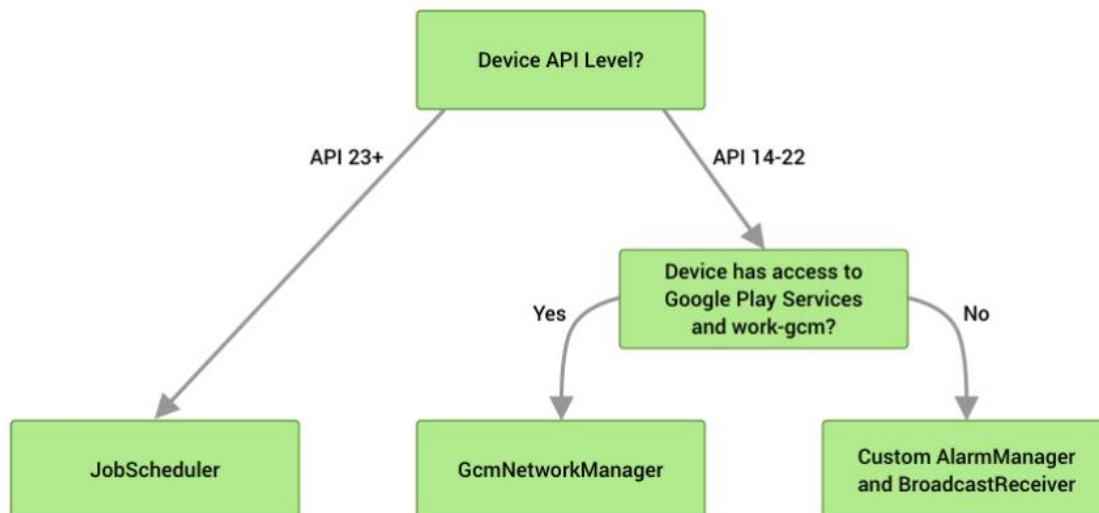
In this part, we will activate a JobService from a broadcast receiver. We will use the last receiver we used for the AlarmManager. **Our goal for this part** is to activate the broadcast receiver, which will trigger the JobScheduler. The job would be downloading something off the web, and then changing the content of the notification to whatever it downloaded. We will simulate the download with a delay function.

WorkManager

As we have seen, There are multiple options to set tasks in the future

WorkManager is an API for reliable deferred background work. By using `enqueue()` method, we can set a `Worker` with a `WorkRequest`, and the WorkManager will take care of almost everything for us. In order to implement that, it uses the JobScheduler and the AlarmManager with the help of broadcast receivers. It also uses the Firebase JobDispatcher, which isn't covered in this document.

WorkManager will automatically choose the appropriate method to run our task, alongside with any constraints we give it. It also takes into consideration the device API level. By doing so, it abstracts all the complexity of writing **deferrable guaranteed work**, and provides us with a clean API that makes it easy to schedule reliable tasks.



This way we manage to cover devices that runs old Android versions, and devices that don't have access to Google Play services (some weird Chinese devices)

Workmanager can schedule both single and periodic tasks. We can also chain tasks together in sequence, where one task runs after the previous task.

Appropriate use cases

There are many different scenarios in which we would like to enqueue a task to be done in the future, or simply under specific conditions. The common thing among them all is that we want to make sure they finish in a finite time, and actually run when needed to.

Deferred – we need to consider if it makes sense for the task to be run at a later time. If the answer is yes, then the task is deferrable. A few examples would be:

- Communicating with a server
- Writing Log files
- Periodically syncing data

Determinism – we need to consider if our task needs to be eventually completed, even if the app is closed. For example, if our app needs to receive critical data from a server, we will probably want it to be done even after the user exits the app.

So, as a rule of thumb, use The WorkManager when you need to do reliable deferred work.(even after user exits the app). **It is not** intended for tasks that needs to run immediately, or at an exact time. For that, we can use the AlarmManager.

Basics

There are a few classes we need to cover before diving into the code:

- **Worker** – the class for the actual task we need to run in the background. We extend this class, and override the `doWork()` method. Since this is a Kotlin tutorial, we will use the [CoroutineWorker](#) class with the `suspend doWork()` method.
- **WorkRequest** – a request to do some work. In here we will also set the constraints on the worker.
- **WorkManager** – The actual class used for scheduling our WorkRequest. This class handle all the resource-saving logic while following our constraints.
- **WorkInfo** – This class contains data about the on going work. We can even observe a LiveData that will wrap this object.

All background work is given a limit of 10 minutes to finish it's execution.

In this section, we will enqueue a `WorkRequest` to fetch the number of corona cases in each city. **We will constraint the worker to work only when the device is charging.** the returned fake data is a type of `HashMap<String, Int>`.

The following dependency is already inside the project:

```
// WorkManager dependency  
implementation "androidx.work:work-runtime-ktx:2.7.0"
```

The project contains initial code for this section in packages:

- `package com.eran.applicationcomponents.data.utils.workmanager`
- `package com.eran.applicationcomponents.data.network`

We will begin by going through some important parts of the initial code

```
suspend fun fetch(context: Context): Boolean {  
  
    // let the user know we started the job  
    AppUtils.notify(context, title: "Working", "Fetching corona statistics")  
  
    // actually get the data  
    val coronaMap = FakeApi.NetworkAccess.getStatistics()  
  
    // work on fetched data  
    val max = coronaMap.maxByOrNull { it.value }  
    max?.let { it: Map.Entry<String, Int>  
        // display results  
        AppUtils.notify(  
            context,  
            "Results",  
            msg: "${max.key} ${"has the most corona cases"} ${max.value}"  
        )  
        return true // indicate success  
    }  
    return false // indicate failure  
}
```

This function manages the call to the outside world, and also let the user know what is going on by using notifications. it fetches the data, **finds the city with the most corona cases**, display the results in a notification and returns a Boolean to indicate success or failure. It's inside the `com.eran.applicationcomponents.data.utils.workmanager` package, `WorkerUtils.kt` class.

```
class FakeApi {
    object NetworkAccess {
        suspend fun getStatistics(): HashMap<String, Int> {
            delay( timeMillis: 5000)

            return hashMapOf(
                "Jerusalem" to 1764,
                "Tel Aviv" to 444,
                "Haifa" to 207,
                "Holon" to 197,
                "Rishon Le'Zion" to 191,
                "Bat Yam" to 172,
            )
        }
    }
}
```

This class implements the network call to receive the statistics. To simulate the call, we use a `delay()` method. The returned results is a hash map with a String key and Int value. It's inside the `com.eran.applicationcomponents.data.network` package.

Let's start by creating a Worker class that will inherit from the `CoroutineWorker`:

```
class CoronaWorker(appContext: Context, workerParams: WorkerParameters) :
    CoroutineWorker(appContext, workerParams) {
}
```

We have an error because we need to implement the `doWork()` method so let's do that now:

```
class CoronaWorker(appContext: Context, workerParams: WorkerParameters) :
    CoroutineWorker(appContext, workerParams) {

    override suspend fun doWork(): Result {
        TODO( reason: "Not yet implemented")
    }
}
```

Notice the `doWork()` method is a suspend method. The default dispatcher is `Dispatchers.Default`. The `coroutineWorker` also handles canceling the coroutine if needed. We will implement this method later on.

We need to provide our worker with a method that will use the `fetch()` method. We also want to give it a timeout incase something will go wrong with the network request:

```
private suspend fun getCoronaData(context: Context): Boolean {
    val utils = WorkerUtils()
    var isValid: Boolean
    try {
        withTimeout( timeMillis: 2000) { this: CoroutineScope
            isValid = utils.fetch(context)
        }
    } catch (e: TimeoutCancellationException) {
        AppUtils.notify(
            context,
            context.getString(R.string.work_cancelled),
            context.getString(R.string.something_wrong)
        )
        return false
    }
    return isValid
}
```

This function returns a Boolean that indicates whether the task has finished successfully or not. It uses `withTimeout()` method that runs a suspending block inside a coroutine with respect to its `timeMillies` parameter. If the time is up, a `TimeoutCancellationException` will be thrown. We give our worker 2 seconds to complete the network request.

Let's get back to the `doWork()` method. This method must return a `Result` object.

The `Result` returned from `doWork()` informs the `WorkManager` service whether the work succeeded and, in the case of failure, whether or not the work should be retried.

- `Result.success()`: The work finished successfully.
- `Result.failure()`: The work failed.
- `Result.retry()`: The work failed and should be tried at another time according to its [retry policy](#).

`Result.failure()` – If there are any workers that are **chained** to the current one, they will all be cancelled. So if they are dependent on the previous worker we should return a `Result.failure()` so they won't try to run.

Implement the `doWork()` method as followed:

```
override suspend fun doWork(): Result {
    val isValid = getCoronaData(applicationContext)
    return if (isValid) {
        Result.success()
    } else {
        Result.failure()
    }
}
```

We call the `getCoronaData()` method, and return success or failure according to the Boolean we got from `getCoronaData()`

In the `AlarmManagerReceiver.kt` class, change the `onReceive()` method to the following:

```
override fun onReceive(context: Context?, intent: Intent?) {

    // set the constraints
    val dataConstraints = Constraints.Builder().setRequiresCharging(true).build()

    // prepare request
    val dataRequest =
        OneTimeWorkRequestBuilder<CoronaWorker>().setConstraints(dataConstraints)
            .build()

    WorkManager.getInstance(context!!).enqueue(dataRequest)
}
```

AlarmManagerReceiver.kt

Set a constraint, prepare a dataRequest and enqueue it. It's that simple.

Note: we can give a worker some initial data to work with by using the `setInputData()` method which accepts a key-value Data object. For example:

```
// prepare request
val dataRequest =
    OneTimeWorkRequestBuilder<CoronaWorker>().setConstraints(dataConstraints)
        .setInputData(workDataOf( ...pairs: "data" to 1337))
        .build()
```

And get it inside the `doWork()` method (must provide a default value as well):

```
override suspend fun doWork(): Result {

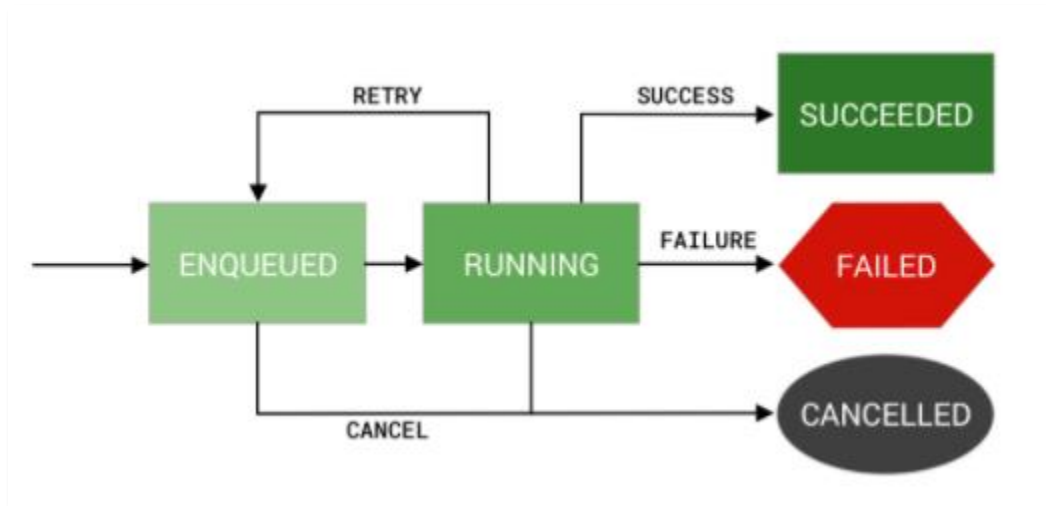
    val number = inputData.getInt( key: "data", defaultValue: 0)

    val isResultValid = getCoronaData(applicationContext)
    return if (isResultValid) {
        Result.success()
    } else {
        Result.failure()
    }
}
```

In this example we set up a [OneTimeWorkRequest](#), which as the name suggests, will be a one-shot operation, and will not repeat itself, unless stated with a [Result.retry\(\)](#). We set a Charging constraint and then pass it to the WorkManager. We can also set an initial delay with the `setInitialDelay()` method.

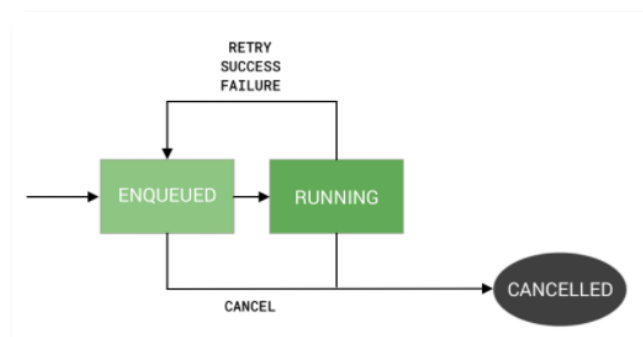
OneTimeWorkRequest States

The work begins in the `ENQUEUED` state. In this state, the work may run as long as it's constraints are met. The next state is `RUNNING` and from there it will either be `SUCCEEDED` or `FAILED`, or `ENQUEUED` again if we set the work to retry in case of failing.



Periodic Work State

With WorkManager we can also create [PeriodicWorkRequest](#) which will be triggered every X amount of time (explained later). The PeriodicWorkRequest does not have a `SUCCEEDED` or `FAILED` state, because it's work doesn't end. It only has an `ENQUEUED` and `CANCELLED` state.



For now, we will run the app. **Because the phone is charging while connected to the computer**, we expect to reach the AlarmManager fragment, hit the alarm, and after 10 seconds see a notification saying "Fetching corona statistics". After 5 more seconds we should see a new notification with the results of the computation.

Spoilers: we got a notification saying the Work has been cancelled. This was to show the functionality of the `withTimeout()` method. Let's fix this by giving more time.

```
private suspend fun getCoronaData(context: Context): Boolean {
    val utils = WorkerUtils()
    var isValid: Boolean
    try {
        withTimeout( timeMillis: 20000) { this: CoroutineScope
            isValid = utils.fetch(context)
        }
    } catch (e: TimeoutCancellationException) {
        AppUtils.notify(
            context,
            "Work has been cancelled",
            "Something went wrong",
        )
        return false
    }
    return isValid
}
```

Run the app again while the phone is charging. Try setting the alarm and exiting the app. The notifications will still be displayed. Try even starting the app without the phone charging, hit the alarm and see that nothing happens. After a while. Plug the phone back in and the workRequest will start immediately.

Important! Be careful of duplicate work when enqueueing. In our example, the work is enqueued by a broadcast receiver which is activated via the AlarmManager, so if we click a lot of time on the alarm button, we won't see any duplicates right away because the system doesn't allow spamming alarms. Instead, we will get unexpected behavior.

If we enqueue multiple times the workRequest right away, we would be able to see in the Logcat the duplicates:

```
2021-10-27 03:21:51.392 22730-22776/com.eran.applicationcomponents I/WM-WorkerWrapper: Worker result SUCCESS for Work [ id=85ea7da2-5816-
2021-10-27 03:21:52.058 22730-22752/com.eran.applicationcomponents I/WM-WorkerWrapper: Worker result SUCCESS for Work [ id=bbc6ac59-3f38-
2021-10-27 03:21:52.747 22730-22776/com.eran.applicationcomponents I/WM-WorkerWrapper: Worker result SUCCESS for Work [ id=38fd24cb-dfe8-
```

Avoid work duplication

Use caution when enqueueing work to avoid duplication. For example, an app might try to upload its logs to a backend service every 24 hours. If you aren't careful, you might end up enqueueing the same task many times, even though the job only needs to run once. To achieve this goal, you can schedule the work as **unique work**.

Unique Work

In order to avoid duplications, we can use unique work functions- `enqueueUniqueWork()` and `beginUniqueWork`. By doing so, we can define the behavior of the duplicate:

- **Replace** – replacing existing work with a new.
- **Keep** – keep the existing work and ignore the new.
- **Append** – append the new work to the end of the existing one. This behavior causes the new work to be **chained** to the existing one

Both of the methods mentioned must be provided with the 3 arguments:

- **uniqueWorkName** – a String to identify the workRequest.
- **existingWorkPolicy** – one of the 3 options mentioned above (replace, keep, append)
- **work** – the workRequest.

For example:

```
WorkManager.getInstance(context!!).enqueueUniqueWork(  
    uniqueWorkName: "workID",  
    ExistingWorkPolicy.REPLACE,  
    dataRequest  
)
```

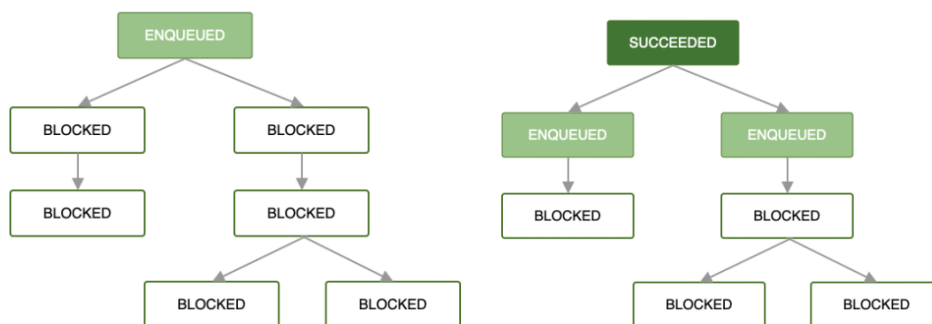
Chaining Work

Next, we are going to write a second worker, and chain his work to the existing worker. WorkManager allows us to chain multiple WorkRequest. To chain two or more WorkRequest, instead of using the `enqueue()` method right away, we start with the `beginWith()` method which accepts a workRequest and returns a [WorkContinuation](#).

The WorkContinuation lets us chain multiple `OneTimeRequest`'s together. Through this object, we can create a complex chain of request. For example, suppose we have 2 chains of requests, and we need for both of them to finish before activating the third chain of requests (by using the `combine(chain1, chain2)` method).

```
WorkContinuation left = workManager.beginWith(A).then(B);
WorkContinuation right = workManager.beginWith(C).then(D);
WorkContinuation final = WorkContinuation.combine(Arrays.asList(left, right)).then(E);
final.enqueue();
```

Chain example



Once the first workRequest finished successfully, the next workRequests may begin. If a workRequest fails, all of its child workRequests are terminated. For now we will keep it simple and chain a second workRequest.

Let's start by creating a finishRequest class that will inherit from CoroutineWorker:

```
class FinishRequest(appContext: Context, workerParams: WorkerParameters) :
    CoroutineWorker(appContext, workerParams) {

    override suspend fun doWork(): Result {
        delay( timeMillis: 3000)
        AppUtils.notify(
            applicationContext,
            applicationContext.getString(R.string.second_worker_finish),
            applicationContext.getString(R.string.work_done),
        )
        return Result.success()
    }
}
```

The strings are already in the project. Return success to let the WorkManager know the worker completed its task.

Once again, Change the `onReceive()` method of the `AlarmManagerReceiver.kt` class

```
override fun onReceive(context: Context, intent: Intent) {

    // set the constraints
    val dataConstraints = Constraints.Builder().setRequiresCharging(true).build()

    // prepare request
    val dataRequest =
        OneTimeWorkRequestBuilder<CoronaWorker>().setConstraints(dataConstraints).build()

    // set the constraints
    val finishConstraints = Constraints.Builder().setRequiresCharging(false).build()

    // prepare request
    val finishRequest =
        OneTimeWorkRequestBuilder<FinishRequest>().setConstraints(finishConstraints).build()

    WorkManager.getInstance(context).beginWith(dataRequest).then(finishRequest).enqueue()

    // enqueue the workRequest
    // WorkManager.getInstance(context).enqueue(dataRequest)
}
```

We set the finishRequest constraint to not require charging, but you can use any constraint you want. Instead of using `enqueue()` right away, we set the WorkManager to begin with a workRequest, followed by another request, and only then using the `enqueue()` method. Run the app and witness magic!

Repeating workRequests

We can also use the `PeriodicWorkRequest` to repeat the work that needs to be done. The first run will execute immediately (or as soon as the conditions are met). The next execution will happen during a time interval that we specify for the `PeriodicWorkRequest`.

The time of execution may be delayed because the `WorkManager` is under battery optimizations obligations. It is intended for use cases where we don't mind inexactness due to battery optimizations. **A `PeriodicWorkRequest` cannot be a part of a chain.** It does not return a `Result` object, and it can only be cancelled explicitly.

```
Kotlin  Java
val saveRequest =
    PeriodicWorkRequestBuilder<SaveImageToFileWorker>(1, TimeUnit.HOURS)
    // Additional configuration
    .build()
```

In the above code, there is a 1 hour interval between calls. **`PeriodicWorkRequest` has a minimum time interval of 15 minutes.** We can even be more flexible and provide a time window inside the workRequest interval: start work in the last 15 minutes of the 1 hour interval.

```
Kotlin  Java
val myUploadWork = PeriodicWorkRequestBuilder<SaveImageToFileWorker>(
    1, TimeUnit.HOURS, // repeatInterval (the period cycle)
    15, TimeUnit.MINUTES) // flexInterval
    .build()
```

Long Running Workers

WorkManager has support for operations that require more than 10 minutes to finish. A signal is provided to the system that if possible, the process shouldn't be terminated. The WorkManager manages and runs a **foreground service** for us, but we need to provide a notification letting the user know a long running service is running.

Inside the `doWork()` method we can call the `setForeground()` method and pass it a **ForegroundInfo** object that contains a notification and its ID. The notification needs to have `setOngoing()` to true so the user won't be able to dismiss the notification. So we must take care of cancelling it, either by letting the user cancel it with a button, or cancel it programmatically. Example (notice the `setForeground()`):

```
override suspend fun doWork(): Result {
    val inputUrl = inputData.getString(KEY_INPUT_URL)
        ?: return Result.failure()
    val outputFile = inputData.getString(KEY_OUTPUT_FILE_NAME)
        ?: return Result.failure()

    // Mark the Worker as important
    val progress = "Starting Download"
    setForeground(createForegroundInfo(progress))
    download(inputUrl, outputFile)
    return Result.success()
}
```


Android 10 and 11 Foreground - Worker Requirements – location, camera and microphone

If the foreground worker needs to have access to location, camera or microphone, we need to declare these [service types](#) in the manifest and at runtime.

In Android 10 and above we need to declare the `location` foreground service type. If the app targets Android 11 and above, we also need to declare the `camera` and `microphone` foreground service type.

When we start a foreground type work at runtime, we get the capability to access the services that we declare in the manifest: the location, camera and microphone.

```
<service
  android:name="androidx.work.impl.foreground.SystemForegroundService"
  android:foregroundServiceType="location|microphone"
  tools:node="merge" />
```

In runtime, we need to specify the service type we need for our service. We don't have to use all of the declared service type in the manifest. For example:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
    setForeground(
        ForegroundInfo(
            notificationId: 1234,
            notification,
            FOREGROUND_SERVICE_TYPE_LOCATION))
}
```

In here, we check if the app is running Android 10 or higher. If it does, we need to use the `setForeground()` method with the `FOREGROUND_SERVICE_TYPE_LOCATION`. The `ForegroundInfo` object takes a notification ID, the notification itself, and the needed service type.

These requirements are meant to protect the users privacy by limiting foreground services that were started from the background. Even though adding foreground service types give a foreground service the capability to access location, camera and microphone, it is still under the access restriction that were introduced in Android 11: If the app is running in the background, the app must be granted with the [ACCESS_BACKGROUND_LOCATION](#) permission, and the foreground service cannot

access the microphone or the camera. But there are a few exemptions from these restrictions such as when the service is started by a system component or by user interaction with a notification.

For the full list of exemptions, read here:

<https://developer.android.com/guide/components/foreground-services#bg-access-restriction-exemptions>

We will talk more about services and their nature in the next section.

Summary

- Used for work that can be deferred.
- WorkManager chooses the appropriate way to run your task based on such factors as the device API level and the app state.
- WorkManager might use a JobScheduler, Firebase JobDispatcher, or AlarmManager with broadcast receivers.
- Guarantees task execution, even if the app or device restarts
- Use the CoroutineWorker to work inside a suspend function.

Let's write the necessary code to implement the JobScheduler. First, let's create the JobService that will be passed to the scheduler. Inside `data.services` package create a new class: DownloadService. This class needs to implement the CoroutineScope interface. Also, we need to inherit from the JobService() class, and override the `onStartJob()` and `onStopJob()` method

```
class DownloadService : CoroutineScope {  
  
    private val job = Job()  
    override val coroutineContext: CoroutineContext  
        get() = job + Dispatchers.IO  
  
}
```

`onStartJob()` – this function is called when the conditions we've set in the JobInfo are met.

Return false if job is already finished.

Return true if the job needs to continue to run after the method returned.

`onStopJob()` – this function is called if the Android system decides to stop execution of the job. For example, if we set our job to work only when the phone is charging, and the job has begun, then if we disconnect the phone, `onStopJob` will be called.

Return false to end the job completely.

Return true if we want to reschedule the job (must be configured in the **JobInfo** using the **Builder**)

Next, we need to inherit from the JobService() class, and override the `onStartJob()` and `onStopJob()` method

First, we notify that downloading has begun. Let's simulate the downloading by launching a coroutine and delaying for 5 seconds, and then notify again that the job is finished. Inside the `onStartJob()` method. We also need to return true because the job isn't finished as soon as the `onStartJob()` method returns (`onStartJob` isn't a suspend function!).

```
override fun onStartJob(p0: JobParameters?): Boolean {  
  
    // notify we starting to download  
    AppUtils.notify(applicationContext, title: "Network", msg: "Downloading...")  
  
    // simulate downloading  
    launch { this: CoroutineScope  
        delay( timeMillis: 5000)  
        AppUtils.notify(applicationContext, title: "Network", msg: "Download Finished")  
        jobFinished(p0, wantsReschedule: false)  
    }  
  
    // Returning true because job isn't finished yet  
    return true  
}
```

jobFinished – call this function to let the JobScheduler know that the job is finished. This function is needed incase our job is asynchronous, and isn't finished when `onStartJob()` returns.

In the `onStopJob()` method we simply return false because we don't need to reschedule our job.

```
override fun onStopJob(p0: JobParameters?): Boolean {  
  
    // no reschedule needed  
    return false  
}
```

We also **MUST** declare our JobService in the manifest:

```
<service  
    android:name=".data.services.DownloadService"  
    android:permission="android.permission.BIND_JOB_SERVICE" />
```

The `android.permission.BIND_JOB_SERVICE` must be provided. If not, this service will be ignored. This permission is defined with the **signature protection level**. A signature protection level means that we will be able to talk to different components of the Android system only if we have the same signature as them. This means that only the JobScheduler is able to run the JobService.

Now, inside `data.utils` package, create a `Scheduler.kt` class. Inside it make a companion object with a `schedule()` method, and a constant that will represent the job id:

```
class Scheduler {
    companion object {
        const val JOB_ID = 100000

        fun schedule(context: Context){

        }
    }
}
```

Inside the `schedule()` method, that's where we provide our `DownloadService.kt` class to the `JobInfo` Builder, and **set our constraints on the job**. Add the following lines. You can also experiment with different kind of constraints.

```
class Scheduler {
    companion object {
        const val JOB_ID = 1000

        fun schedule(context: Context) {
            val serviceComponent = ComponentName(context, DownloadService::class.java)
            val builder = JobInfo.Builder(JOB_ID, serviceComponent)

            //set constraints
            builder.setRequiresCharging(true)

            val jobScheduler = context.getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler
            jobScheduler.schedule(builder.build());
        }
    }
}
```

We can also add multiple constraints:

```
//set constraints
builder
    .setRequiredNetworkType(JobInfo.NETWORK_TYPE_CELLULAR)
    .setRequiresCharging(true)
    .setRequiresDeviceIdle(true)
    .setRequiresStorageNotLow(true)
```

- A constraint for requiring the device to be with cellular network (requires SDK 28).
- A constraint for requiring the device to be in idle mode
- A constraint for requiring the device to have enough storage (the point where the user receives a “low warning” message from the system) – requires Android 8.0+

For a full list of the available constraints:

<https://developer.android.com/reference/android/app/job/JobInfo.Builder#summary>

`JobScheduler.schedule()` – the function to schedule a job. If there’s already a scheduled job with the same ID, the old job will be replaced with the new one. If an existing job with the same ID is already running, it will be stopped when we schedule with the same ID.

Now, Change the `onReceive()` function in `AlarmManagerReceiver`:

```
class AlarmManagerReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        Scheduler.schedule(context!!)
        // AppUtils.notify(
        //     context!!,
        //     "My notification",
        //     "Hi there!"
        // )
    }
}
```

We are now ready to try our `JobScheduler`. Unplug your phone from the charger, and start the alarm. Because we set `requiresCharging` to true, nothing will happen. Plug the phone back in and the task will be executed immediately.

Notes:

- To use the `JobScheduler`, we need a **`JobInfo.Builder`** to configure the constraints and conditions, and a **`JobService`** to implement the job.
- The `JobService` must be declared with a `BIND_JOB_SERVICE` permission in the manifest.
- `onStartJob()` returns a Boolean that indicates if the job needs more time to be finished (such as asynchronous action). If return true, we must call `jobFinished()`.
- `JobService` works on the main thread. For any long operations and networking, use a different thread.
- `JobScheduler` groups tasks together to save system resources, so we don't have full control of when job gets executed.

Overview

Android service is one of the 4 fundamental application components. It's intended to run long operations, even after the user exited the app. The service does not have a user interface. That means that it doesn't contain any view references at all. **It does not mean we can't display it's data on the UI.** We can interact with it via broadcasts, and even bind our app's activity to it, and get references to its public methods and variables. We can even set LiveData objects inside it and observe them while the app is alive. More on communication later on. Let's take a look at the basics:

Lifecycle

The lifecycle of a service is much simpler than that of an activity. However, it's even more important that you pay close attention to how your service is created and destroyed because a service can run in the background without the user being aware.

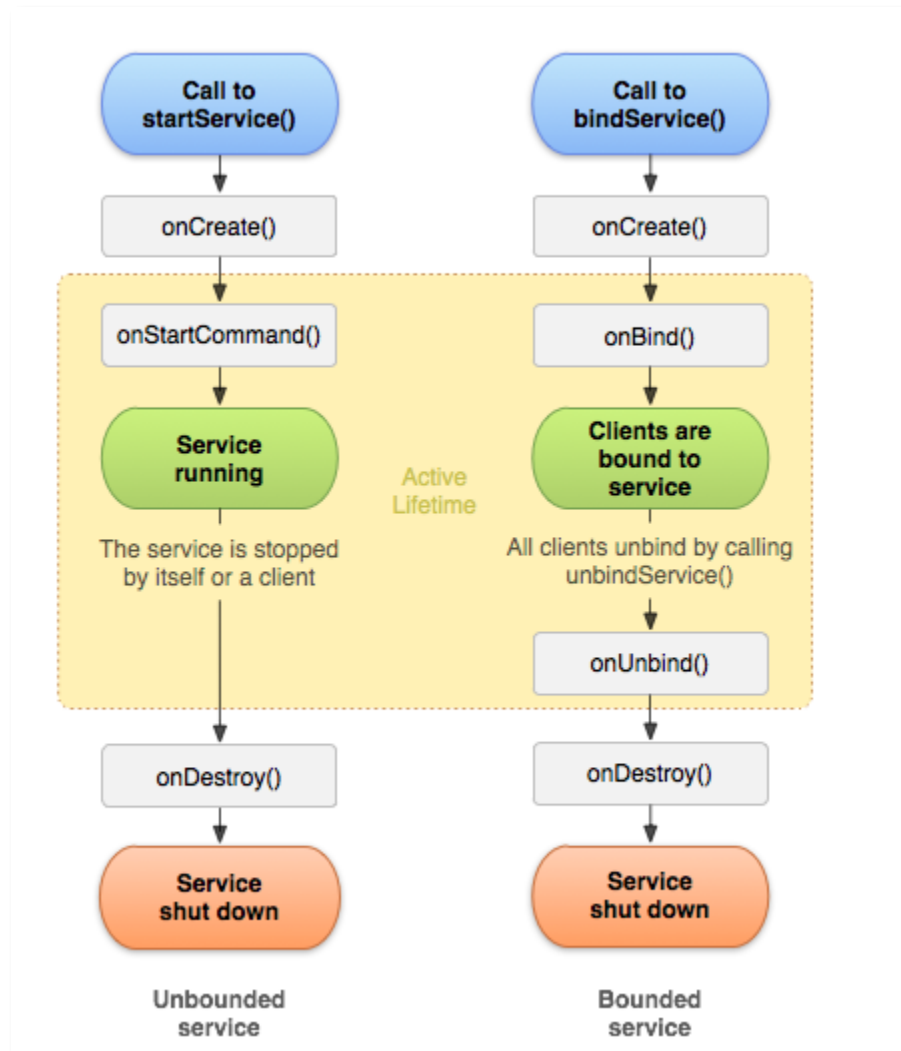
A services lifecycle can have two forms:

Started – the service is started by calling the `startService()` method. From there, the server runs until it is stopped by calling the `stopSelf()` method. It can also be stopped by a different component by calling `stopService()`. When it gets to "stopped", the system kills it.

Bound – the service begins its lifecycle when another component binds to it by calling the `bindService()` method. By using this bound service, we can communicate with it via an `IBinder` interface, which will enable us to get a reference to it. Multiple components can bound to a single service, and can unbind by calling `unbindService()` method. **Only when** the last component bound to the service calls this method, the service dies (the other option is it simply crashed). It does not need to stop itself .

We can also mix these two forms. We can bind to a service that had already started. An implementation for that would be a music player: the users might want to do other things in their device while listening to music, and then jump back to the music player to take wider control of the app and change a song, or skip to a different part of the song.

Note: The system may stop the service by itself when it is running low on memory, and must regain some. There is an option of setting the service as a **Foreground** service, which lets the service stay alive (almost always) despite the memory hungry system. But if the system does decide to terminate the service, it will restart it when possible. More on Foreground services later on.



Basic methods of the lifecycle

onStartCommand() – This method is called when a component requests to start the service. The system invokes this method by calling `startService()`. The service is then started and can run indefinitely. It is our job to make sure it is stopped when needed to either by calling `stopSelf()` from inside the service, or `stopService()` from outside of it.

onBind() – This method is called when another component request to bind with the service by calling `bindService()`. For binding with a service, we need to have an interface for the binding component to use in order to communicate with the service.

Note: only activities, other services and content providers can bind to a service. We cannot bind to a service from a broadcast receiver. Unless it is registered dynamically because then it's lifetime is tied to another live component.

[reference](#)

onCreate() – The system calls this method when the service is created. This method will be run once per lifecycle to setup the service. Meaning, if the service is already running, this method will not be called again.

onDestory() – This method is called when the service is no longer in use. In this method, we need to clean up any resources we might have used such as thread and coroutines, listeners and broadcast receivers.

onUnbind() – This method is called when all the components that were bound to the service disconnected.

Use cases

The service is intended to run in the “background”, mostly away from the users eyes, when we need to do some tasks that does not involve views. If we simply want to perform operations outside of the main thread, we have many, many ways to do that, as seen in the previous sections.

If we need to make long running operations that needs to be kept alive after the user exited or switched to a different app, then this is a good usage of the service component. In general, if the user doesn't need to interact with the application, but the app must still be alive, then a service is a good option.

Declaring a service in the manifest

We must declare all of our services in the app's manifest, as shown before with the JobService for example. To declare a service, we add the `<service>` tag in the `<application>` part of the manifest. Example:

```
<manifest ... >
  ...
  <application ... >
    <service android:name=".ExampleService" />
    ...
  </application>
</manifest>
```

The `name` attribute is the only required attribute. It is the name of our service class.

There are more attributes:

- `exported` – whether or not other apps can start our service.
- `enabled` – whether or not the service can be started by the system.
- `permission` – specify a permission that a component must have in order to start the service or bind to it.

for a full list of attributes, visit:

<https://developer.android.com/guide/topics/manifest/service-element#prmsn>

Creating a started service

Let's begin with something basic. In the `data.services` package, create a new class, and inherit from the `Service` class. Call it `MyService`:

```
class MyService : Service() {  
}
```

The compiler gives an error because we need to override the `onBind()` method. So let's override this method. Also, make the class implement the `CoroutineScope` so we would be able to launch coroutines from the service:

```
class MyService : Service(), CoroutineScope {  
    override fun onBind(p0: Intent?): IBinder? {  
        TODO(reason: "Not yet implemented")  
    }  
  
    override val coroutineContext: CoroutineContext  
        get() = TODO(reason: "Not yet implemented")  
}
```

For now, in the `onBind()` method we will return null. By returning null we deny any binding to our service. And for the `coroutineContext`, we will use the `Dispatchers.DEFAULT`:

```
class MyService : Service(), CoroutineScope {  
  
    override fun onBind(p0: Intent?): IBinder? {  
        return null  
    }  
  
    private val job = Job()  
    override val coroutineContext: CoroutineContext  
        get() = job + Dispatchers.Default  
}
```

Next, we will override the `onDestroy()` method so we can cancel the coroutine job:

```
override fun onDestroy() {  
    super.onDestroy()  
    job.cancel()  
}
```

And now for the real work. Override the `onStartCommand()` method of the service. As mentioned before, when we start the service with the `startService()` method, the `onStartCommand()` will be invoked and the server will be created and started.

```
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  
    return super.onStartCommand(intent, flags, startId)  
}
```

Explanation:

This method receives an intent that we pass in the `startService()` method.

The flags can be either:

- `START_FLAG_REDELIVERY` – In case the system killed the service, the previously delivered intent will be redelivered.
- `START_FLAG_RETRY` – This flag means that the service was restarted because the last call to `onStartCommand()` never returned **before** the service was destroyed. It was either killed by the system before it returned, or simply crashed.

- START_STICKY – if the system kills the service **after** onStartCommand() returns, the service will be recreated and call onStartCommand() again, but instead of redelivering the last intent. It will deliver null.

Note: In case we don't want the service to start after the system kills it, return START_NOT_STICKY in onStartCommand() to indicate to not recreate the service unless there are pending intents to deliver.

For more flags, visit here:

https://developer.android.com/reference/android/app/Service#constants_1

startId parameter – this is a unique integer that represents the specific request to start the service

Let's continue with our code. In the onStartCommand() method, use the AppUtils.notify() method to create a notification with any text you want. You can also use the preexisting strings:

```
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    AppUtils.notify(
        applicationContext,
        getString(R.string.my_service),
        getString(R.string.hello)
    )
    return super.onStartCommand(intent, flags, startId)
}
```

Let's also put a toast in the onCreate() method of the service, so we can see that it only gets called once per service creation:

```
override fun onCreate() {
    super.onCreate()
    AppUtils.makeToast(context: this, getString(R.string.service_created))
}
```

When we want to pass messages (intents) to an already started service, we also use the `onStartCommand()` method, but `onCreate()` will not trigger because the service had already been created. In the intent we pass, we can plant the data we want to give to the receiver, and we get it by getting the intents EXTRA's. We will demonstrate this by sending 2 intents one after the other later on.

We also must declare the service in the manifest:

```
<service android:name=".data.services.MyService"/>
```

Since this is considered as a local service (this is our service, started by our app), there is no need to set the `exported` attribute, as we did with the `BluetoothStateReceiver`.

In the service `onDestroy`, create a toast message to indicate the service was destroyed.

```
override fun onDestroy() {  
    super.onDestroy()  
    AppUtils.makeToast(context: this, getString(R.string.service_destroyed))  
    job.cancel()  
}
```

Next, let's work on the `ServiceFragment.kt` already present in the project. This fragment contains two buttons: one for starting the service and one for binding with it. We start with a regular `startService()` method.

Inside the `onViewCreated()` method of the `ServiceFragment`, use the `binding` object to set an `OnClickListener` for the `startService` button. From there, we will start the service using the activity's context, and pass an Intent with the context, and our Service `class.java`.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
    super.onViewCreated(view, savedInstanceState)  
  
    binding.startService.setOnClickListener { it: View!  
        requireActivity().startService(Intent(requireActivity(), MyService::class.java))  
    }  
}
```

Don't forget to stop the service when no longer needed. In this example, we will stop the service from outside, so we need to call the `stopService()` method. We will call this method from the `onDestroyView()` method of the ServiceFragment:

```
override fun onDestroyView() {  
    super.onDestroyView()  
    requireActivity().stopService(Intent(requireActivity(), MyService::class.java))  
}
```

We are now ready to run the app. Start the app and navigate to the ServiceFragment. Click the start service button, you should see a notification with the title and text you gave it earlier. Also, try to navigate back from the ServiceFragment to see a Toast that the service is destroyed.

Now, Let's send some data to the service through an intent after it started. Write an `onClickListener` for the `startService` button

Create 2 constants for the intent extra:

```
companion object {  
    const val EXTRA_KEY = "service_key"  
    const val DATA_EXTRA = "My custom data for the service"  
}
```

```
binding.startService.setOnLongClickListener { it: View!  
    val intent = Intent(requireActivity(), MyService::class.java)  
    intent.putExtra(EXTRA_KEY, DATA_EXTRA)  
    requireActivity().startService(intent)  
    true ^setOnLongClickListener  
}
```

Return true to consume the long click.

Now in the service class, let's work on the received data. Inside the `onStartCommand()` method, get the extra from the intent parameter. If it matches the `EXTRA_KEY`, get the data and display it in a notification. If the data is null, that means a short click was made so just display the first notification. You may remove the old notification from the `onStartCommand()`


```
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  
    val data: String? = intent?.getStringExtra(EXTRA_KEY)  
    if (data != null) {  
        // long click  
        AppUtils.notify(context: this, getString(R.string.my_service), data)  
    } else {  
        // short click  
        AppUtils.notify(context: this, getString(R.string.my_service), getString(R.string.hello))  
    }  
    return super.onStartCommand(intent, flags, startId)  
}
```

Run the app. You should first see the first notification greeting you Hello, and after you long click the button, you should see the second notification with our data.

Next, we will try to bind to our service, and pass data back to our activity from the service.

Bound Service

When binding a service, we must provide a Binder object that will provide us the interface that we need to communicate freely with the service. We need to extend the Binder class, prepare the Binder object and return it in `onBind()` method instead of null.

```
private val binder = LocalBinder()  
  
inner class LocalBinder : Binder() {  
    fun getService(): MyService = this@MyService  
}  
  
override fun onBind(p0: Intent?): IBinder {  
    return binder  
}
```

Note: return `@MyService` to return the context of the service, and not the LocalBinder.

the Binder object is the current instance of our Service, that will give us access to the services public methods and variables. Lets also add a random number generator to the service class that we will use from the activity.

```
val randomNumber: Int = (0..200).random()
```

Let's get back to the `ServiceFragment.kt` class. Create a service variable of the type of our service:

```
private var myService: MyService? = null
```

Now, create the connection object that will implement the callback to when the service gets bound. Inside the `onServiceConnected()` method, access the random number and show it in a snackbar. Or in a notification... you know the drill.

In the `onServiceDisconnected()`, show a snackbar to indicate disconnection.

```
private val connection = object : ServiceConnection {
    override fun onServiceConnected(className: ComponentName?, service: IBinder?) {
        val binder = service as MyService.LocalBinder
        myService = binder.getService()

        AppUtils.showSnackbar(binding.bindService, myService.randomNumber.toString())
    }

    override fun onServiceDisconnected(p0: ComponentName?) {
        AppUtils.showSnackbar(binding.bindService, getString(R.string.service_disconnect))
    }
}
```

Create an `onClick` listener for the `bindService` button with the following code:

```
binding.bindService.setOnClickListener { it: View!
    val intent = Intent(requireActivity(), MyService::class.java)
    requireActivity().bindService(intent, connection, Context.BIND_AUTO_CREATE)
}
```

In here, we create a simple intent provided with the context and our service class.

Then we use the activity's context to bind the service, passing the intent, the connection variable, and a `BIND_AUTO_CREATE` constant that indicates to automatically create the service if it wasn't already created. Note that if it does create the service, the `onStartCommand()` method will only be called from an explicit call to `startService()`

As mentioned above, a bound service dies only when nothing is bound to it. So in the `onDestroyView()` of the ServiceFragment, we need to unbind in case we were bounded:

```
override fun onDestroyView() {  
    super.onDestroyView()  
    requireActivity().stopService(Intent(requireActivity(), MyService::class.java))  
    myService?.let { it: MyService  
        requireActivity().unbindService(connection)  
    }  
}
```

Run the app. Try first starting the service, and then binding to it. At first, you should see the greeting notification, and after binding, you should see the random number. Exit the fragment and enter again in order to destroy the service. Try the other way around and first bind with the service. Notice that while the random number was fetched, there weren't any notifications. As explained before, `onStartCommand()` only gets called from an explicit `onStartService()`

Foreground services

A foreground service is a service that the user is aware of and isn't a candidate for the system to terminate when it is low on memory. For example, a music player that needs to play music when the user isn't in the app, or a navigation app that needs to display directions while the user is doing something else in their phone.

An app is considered to be in the foreground if it has a visible activity or a service that is in the foreground mode.

The foreground service must display a notification for the user, indicating that a foreground service is running. **As long as the foreground service is running, the notification must not be dismissed.** To implement that, we will create another notify method that will appropriately set up the notification so it can't be dismissed. Through this notification, you can also provide the user some UI to control what is going on. For example:

Note: Apps that target Android 9 (API level 28) or higher and use foreground services must request the `FOREGROUND_SERVICE` permission. With this permission, the system will allow our app to run a foreground service.

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

Inside the service class, let's create a `makeNotification()` method. You can use your own text to display in the notification, or use preexisting ones.

```
private fun makeNotification(): Notification =
    NotificationCompat.Builder(context: this, AppUtils.NOTIFICATION_CHANNEL_ID)
        .setContentTitle(getText(R.string.foreground_title))
        .setContentText(getText(R.string.foreground_msg))
        .setSmallIcon(R.drawable.ic_favorite)
        .build()
```

Note: without setting a title, text and an icon we get unexpected behavior. So make sure to put them all in there.

In the `onStartCommand()` method, disable the last code we wrote so it won't bother us.

To help demonstrate the foreground service, we need to disable the `stopService()` method in the `onDestroyView()` method in the ServiceFragment.

```
override fun onDestroyView() {
    super.onDestroyView()
    // requireActivity().stopService(Intent(requireActivity(), MyService::class.java))
    myService?.let { it: MyService
        requireActivity().unbindService(connection)
    }
}
```

we are also going to create a text-to-speech method for our service that will play when the service is created. First, add the following variable:

```
private var tts: TextToSpeech? = null
```

Now, let's initialize the `tts` object in the `onCreate()` method

```
tts = TextToSpeech( context: this) { status ->
    if (status != TextToSpeech.ERROR) {
        tts?.language = Locale.US
        tts?.speak(
            getString(R.string.speech),
            TextToSpeech.QUEUE_FLUSH,
            params: null,
            utteranceId: "UNIQUE ID"
        )
    }
}
```

The first parameter for the `TextToSpeech` is the context. The second one is a listener. The listener is used to indicate our service that the text-to-speech engine is ready to use. At that moment, we will begin the speech.

`speak()` – parameters:

- **text** - String to be spoken.
- **queueMode** - Queueing strategy – In this example, the new entry will replace all existing entries.
- **params** – A Bundle that contains configurations for the speech such as stream type and volume.
- **utteranceId** – A unique string for identifying this speak request.

Use the preexisting string to pass to the `speak()` method, or just enter your own.

Also, we need to know when the text-to-speech is done so we can close it. Add the `UtteranceProgressListener`:

```
tts?.setOnUtteranceProgressListener(object : UtteranceProgressListener() {  
    override fun onStart(p0: String?) {}  
  
    override fun onDone(p0: String?) {  
        tts?.stop()  
        tts?.shutdown()  
        stopSelf()  
    }  
  
    override fun onError(p0: String?) {  
    }  
})
```

Also, we need to shut down the TextToSpeech in case it is currently working, when we destroy the service:

```
override fun onDestroy() {  
    super.onDestroy()  
    AppUtils.makeToast(context: this, getString(R.string.service_destroyed))  
    job.cancel()  
    tts?.let { it: TextToSpeech  
        if (it.isSpeaking) {  
            it.stop()  
            it.shutdown()  
        }  
    }  
}
```

Before we make the service a foreground type, let's try to run the app without it. Start the app and Navigate to the ServiceFragment. When you start the service, a speech will play, but without indication for the user that something is active. While it plays, kill the app. The speech will stop immediately, because the app wasn't in foreground mode the service died with the app.

Now let's make our service a foreground service.

Call `startForeground()` in the `onCreate()` method and pass some unique id (Integer) and the `makeNotification()` method we just created.

```
override fun onCreate() {  
    super.onCreate()  
    startForeground(FOREGROUND_ID, makeNotification())  
    AppUtils.makeToast(context: this, this.getString(R.string.service_created))  
}
```

```
companion object {  
    const val FOREGROUND_ID = 321  
}
```

Run the app again and start the service. While the speech plays, kill the app. Because we moved the service to the foreground, it won't die and keep playing the speech.

Foreground service from the background

Apps that target **Android 8.0** and above cannot start a foreground service from the background. If an app tries to start a foreground service while the app is in the background, we will get a [ForegroundServiceStartedNotAllowedException](#).

Try it for yourself. Disable the code inside the `AlarmManager.kt` `onReceive()` method, and add the following line:

```
context.startService(Intent(context, MyService::class.java))
```

Run the app, enter the AlarmFragment and hit the button. Kill the app and after a few seconds, you will get the `ForegroundServiceStartedNotAlloedException`:

```
Caused by: java.lang.IllegalStateException: Not allowed to start service Intent { cmp=com.eran.applicationcomponents/.data.services.MyService }: app is in background
```

special cases

there are cases where the app can start a foreground service even while the app is running in the background. For example:

- The app receives a high- priority message from the Firebase Cloud Messaging service (not covered in this document).
- The app triggers an exact alarm to complete an action that the user requested.
- The app receives a Bluetooth broadcast.

For a full list, visit:

<https://developer.android.com/guide/components/foreground-services#background-start-restriction-exemptions>

To overcome this background execution limitation, we need to call the `startForegroundService()` method. This method is allowed to be called even while the app is in the background.

Note: the app must call the service's `startForeground()` method within five seconds after the service is created!

Let's go back to the `AlarmManagerReceiver.kt` class, and add the necessary code to properly start a foreground service from the background.

First, disable every line of code in the `onReceive()` method, then add the following lines:

```
val foregroundIntent = Intent(context, MyService::class.java)
ContextCompat.startForegroundService(context!!, foregroundIntent)
```

Now, Let's get back to our service class. Because we are not going to stop the foreground service ourselves through the fragment, We should provide a way for the users to do it by themselves in case they want to. It's not always needed, but it's good to have this option when the service isn't doing something critical.

Add the following constants to the companion object in the service class:

```
const val ACTION_CLOSE = "close_notification"
const val CLOSE_REQUEST_CODE = 11
```

Add the following `createCloseAction()` method:


```
private fun createCloseAction(): NotificationCompat.Action {  
  
    val closeIntent = Intent( packageContext: this, MyService::class.java)  
    closeIntent.setAction(ACTION_CLOSE)  
    val pendingIntent = PendingIntent.getService(  
        context: this,  
        CLOSE_REQUEST_CODE,  
        closeIntent,  
        PendingIntent.FLAG_UPDATE_CURRENT  
    )  
  
    return NotificationCompat.Action.Builder(  
        R.drawable.ic_exit,  
        getString(R.string.close),  
        pendingIntent  
    ).build()  
}
```

This method returns a `NotificationCompat.Action` object.

Put this object inside the `makeNotification()` method as followed:

```
private fun makeNotification(): Notification =  
    NotificationCompat.Builder( context: this, AppUtils.NOTIFICATION_CHANNEL_ID)  
        .setContentTitle(getText(R.string.foreground_title))  
        .setContentText(getText(R.string.foreground_msg))  
        .setSmallIcon(R.drawable.ic_favorite)  
        .addAction(createCloseAction())  
        .build()
```

By adding this action to the notification, we create an exit button in the notification, and the users may now stop the foreground service by their own.

Now we need to handle the action in the service class:

```
override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
    if (intent?.action.equals(ACTION_CLOSE)) {
        stopSelf()
    }
    val data: String? = intent?.getStringExtra(ServiceFragment.EXTRA_KEY)
    if (data!=null){
        AppUtils.notify(this, getString(R.string.my_service), data)
    }else{
        AppUtils.notify(this,getString(R.string.my_service), getString(R.string.hello))
    }

    return super.onStartCommand(intent, flags, startId)
}
```

Run the app and start the service, our notification now has an exit button. click on the button and the service should be destroyed with toast message saying the service was destroyed.

Summary

- A service runs by default on the main thread.
- Must inherit from the Service class
- Communicate back and fourth through intents or by binding.
- Apps that target Android 9.0 and higher must have the FOREGROUND_SERVICE permission.
- Must declare the service in the manifest.
- Must provide a notification when setting a service to run in foreground.
- Must use the startForegroundService() if starting a foreground service from the background.
- Binding with the service will not call the `onStartCommand()` method.
- Each broadcast to the service will go through the `onStartCommand()` method. From there we can get the data from the intent.
- Consider working with the WorkManager API for cleaner, and better control over background work.