Advanced Kotlin

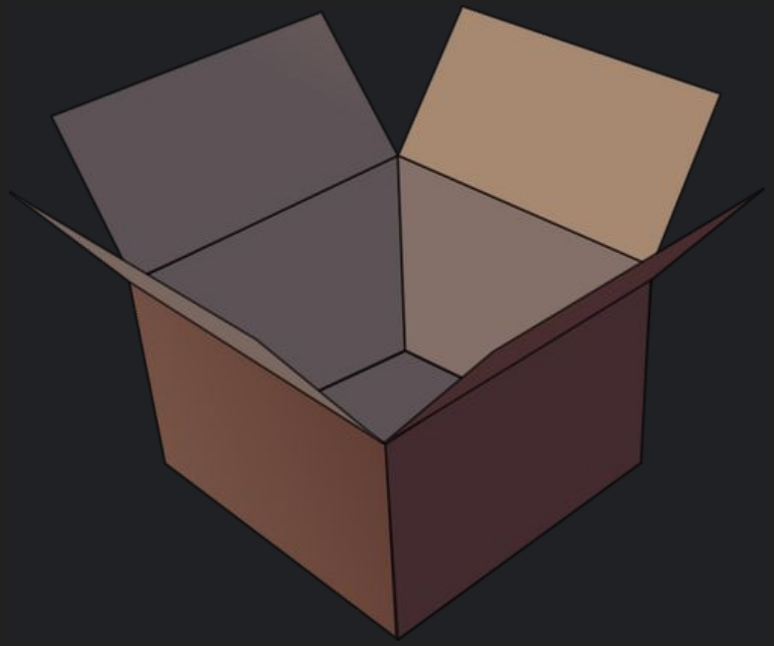
Continue with nullability, collections, generics, complex functions and delegates

Luka Kordić
Android @Spyrosoft
GDG Osijek

- Nullability
- Delegates
- Lambda & higher order functions
- Polymorphism & Abstractions
- Override & overload
- Collections
- Types of classes

Nullability

Absence of value



Null - "The Billion-Dolar Mistake"

Java code - don’t type

```java
public class Person{
    String name;


 public static void main(String[] args) {
    System.out.println(new Person().name.length());
 }
```

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "String.length()" because "name" is null
    at Person.main(Person.java:5)
```

Nullability

- Built into the type system
- Nullable type definition using “Type?” syntax

```
val name: String = null
```

Null can not be a value of a non-null type String

Add 'toString()' call Alt+Shift+Enter More actions... Alt+Enter

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

Null handling

Null handling

- **Safe-call operator ?.**

```
val nameLength = name?.length
println(nameLength) //output: null
```

Null handling

- Null-check with if

```
if (name != null) println(name.length) else println(0)
```

Null handling

- Elvis operator ?:

```
val nameLengthElvis = name?.length ?: 0
println(nameLengthElvis)  //output: 0
```

Null handling

- **Not-null assertion operator !!**

```
val name: String? = null

println(name!!.length)
```

Platform type

Additional resources

Calling Java from Kotlin | Kotlin Documentation

Null safety | Kotlin Documentation

Nulls and Null Safety - Dave Leeds on Kotlin

Inheritance

Inheritance

- A superclass shares properties and methods with a subclass
- A subclass can only inherit from one superclass
- Kotlin classes are final by default

```kotlin
open class Person(
    val name: String,
    val lastName: String
)
```

```kotlin
class Teacher(
    name: String,
    lastName: String,
    salary: Double
) : Worker(name, lastName,
Job("Teacher", salary))
```

```kotlin
open class Worker(
    name: String,
    lastName: String,
    val job: Job
) : Person(name, lastName)
```

```
open class Person(
    val name: String,
    val lastName: String
)
```

```
class Teacher(
    name: String,
    lastName: String,
    salary: Double
) : Worker(name, lastName,
Job("Teacher", salary))
```

```
open class Worker(
    name: String,
    lastName: String,
    val job: Job
) : Person(name, lastName)
```

```kotlin
open class Person(
    val name: String,
    val lastName: String
)


open class Worker(
    name: String,
    lastName: String,
    val job: Job
) : Person(name, lastName)
```

```kotlin
class Teacher(
    name: String,
    lastName: String,
    salary: Double
) : Worker(name, lastName,
Job("Teacher", salary))
```

Inheritance

- Open keyword - classes are closed by default
- Inheritance creates a class hierarchy
- Only non-private properties and methods are transferred
- protected - limits visibility only to subclasses

Interfaces

Interfaces

- Similar to classes, but always abstract
- “Contract” you have to follow
- Abstract methods must be implemented
- Any number of interfaces can be implemented

```kotlin
interface OnClickListener {
    fun onClick()
}

interface OnLongClickListener {
    fun onLongClick()
}

interface CombinedClickListener : OnClickListener, OnLongClickListener

class CustomListener: CombinedClickListener {

    override fun onClick() {
        // Implement
    }

    override fun onLongClick() {
        // Implement
    }
}
```

```
interface OnClickListener {
    fun onClick()
}

interface OnLongClickListener {
    fun onLongClick()
}

interface CombinedClickListener : OnClickListener, OnLongClickListener

class CustomListener: CombinedClickListener {

    override fun onClick() {
        // Implement
    }

    override fun onLongClick() {
        // Implement
    }
}
```

```
interface OnClickListener {
    fun onClick()
}

interface OnLongClickListener {
    fun onLongClick()
}

interface CombinedClickListener : OnClickListener, OnLongClickListener

class CustomListener: CombinedClickListener {

    override fun onClick() {
        // Implement
    }

    override fun onLongClick() {
        // Implement
    }
}
```

Abstractions & polymorphism

Abstractions & polymorphism

- Ability of a type to take many forms
- We can use a common (more abstract) type for specific behavior without knowing about the implementations

```kotlin
interface Attacker {
    fun attack()
}

class Swordsman: Attacker {
    override fun attack() {
        println("Swinging a sword...")
    }
}

class Archer: Attacker {
    override fun attack() {
        println("Drawing a bow...")
    }
}

class Dragon: Attacker {
    override fun attack() {
        println("Breathing fire...")
    }
}
```

```kotlin
fun main() {
    val boss = Boss(Archer())
    println(boss.attackPlayer())
}
```

Output:

```
Drawing a bow…
```

Abstractions & polymorphism

- Many design patterns use polymorphism
- Reduces complexity
- Simplifies syntax

Override & overload

Override

- Changes parent's implementation
- override fun functionName()

```
open class Person(
    val name: String,
    val lastName: String
) {
    override fun toString(): String {
        return "My name is $name $lastName"
    }
}
```

```
class Teacher(
    name: String,
    lastName: String,
    salary: Double
) : Worker(name, lastName, Job("Teacher", salary)) {
    override fun toString(): String {
        return "My name is Mister/Miss $lastName"
    }
}
```

```
open class Worker(
    name: String,
    lastName: String,
    protected val job: Job
) : Person(name, lastName) {
    override fun toString(): String {
        return "My name is $name $lastName. I work as a
${job.name}"
    }
}
```

Overload

- Same name
- Different function signature

```kotlin
fun doWork() {
    // Do something
}

fun doWork(assistant: Worker) {
    // Do something with the assistant’s help
}
```

Different number of params

```
fun doWork() {
    // Do something
}

fun doWork(assistant: Worker) {
    // Do something the assistant's help
}
```

Different number of params

```
fun sum(float1: Float, float2: Float): Float {
   return float1 + float2
}

fun sum(int1: Int, int2: Int): Int {
   return int1 + int2
}
```

Different types of params

Additional resources

https://kotlinlang.org/docs/inheritance.html

https://typealias.com/start/kotlin-abstract-and-open-classes/

https://typealias.com/start/kotlin-interfaces/

Delegation

Delegation

Class behavior delegation

Delegated properties

Class behavior delegation

```
interface Engineer {
    fun buildApp()
    fun testApp()
    fun writeNewBackendApi()
}
```

Class behavior delegation

```
interface Engineer {
    fun buildApp()
    fun testApp()
    fun writeNewBackendApi()
}


class SeniorEngineer : Engineer

class Manager : Engineer
```

```kotlin
class SeniorEngineer : Engineer {

    override fun buildApp() {
        println("I'm building a new Android app!")
    }

    override fun testApp() {
        println("No bugs found! Ready to launch...")
    }

    override fun writeNewBackendApi() {
        println("Creating user endpoints.")
    }
}
```

```kotlin
class Manager(engineer: Engineer) : Engineer by engineer {
    fun organizeMeeting() {
        println("Let's have a sprint planning!")
    }
}
```

```kotlin
val filip = SeniorEngineer()
val pero = Manager(engineer = filip)
pero.organizeMeeting()
pero.buildApp()
```

Delegated properties

- Lazy properties
  - Value computed on the first access

```
val expensiveObject by lazy { produceExpensiveObject() }
```

Delegated properties

- Observable properties
  - Notify when changes occur

```
var name by Delegates.observable("Filip") { _, oldValue,
newValue ->
   println("Old: $oldValue, new: $newValue")
}
```

```
var name by Delegates.observable("Filip") { _, oldValue, newValue ->
   println("Old: $oldValue, new: $newValue")
}
name = "Luka"
```

Output:

Old: Filip, new: Luka

Delegated properties

- Map
  - Store properties in a Map

```
val nameToJob = mapOf(
    "filip" to "Compose Dev",
    "luka" to "Android Dev"
)

val filip by nameToJob
println(filip)
```

Delegated properties

- Map
  - Store properties in a Map

```
val nameToJob = mapOf(
    "filip" to "Compose Dev",
    "luka" to "Android Dev"
)

val filip by nameToJob
println(filip)
```

Output: Compose Dev

Lateinit

- Initialize a var at a later point in time
- We can use it in the following cases:
  - Property defined within the class
  - No custom get/set accessors
  - Local & top-level variables
  - Non-null, non-primitive type

```
lateinit var course: Course

if(::course.isInitialized) {
  println(course.name)
}
```

Additional resources

https://typealias.com/start/kotlin-delegation/

https://kotlinlang.org/docs/delegation.html#delegation.md

Higher-order functions

Higher-order functions

- Can be stored in variables
- Can be sent as arguments to other functions
- Can be returned from a function
- Function types

Function types

- Type with no params and Unit return type

```
val noParamFunType: () -> Unit
```

Function types

- Type with String param and Unit return type

```
val oneParamFunType: (String) -> Unit
```

Function types

- Type with String param and String return type

```
val funWithReturnType: (String) -> String
```

Function types

- Type with receiver

```
val containsVowels: String.() -> Boolean = {
    this.contains("a", true)
      || this.contains("e", true)
      || this.contains("i", true)
      || this.contains("o", true)
      || this.contains("u", true)
}
```

Function types

- Type with receiver

```
val containsVowels: String.() -> Boolean = {
    this.contains("a", true)
      || this.contains("e", true)
      || this.contains("i", true)
      || this.contains("o", true)
      || this.contains("u", true)
}
```

Function literals

- functions that are not declared but are passed immediately as an expression
- Values for function types
  - Lambda literals `{ a, b -> a + b }`
  - Anonymous function `fun(a, b): Int = a - b`

Function type

- `() -> Unit`
- `(String) -> Boolean`
- `(Int, Int) -> Int`
- `String.() -> Boolean`

Function literal

- `{ }`
- `{ it.contains('a') }`
- `{ a, b -> a + b }`
- `{ this.contains('a') }`

Lambdas have no return keyword, the last expression is returned.

Higher-order functions usage

- Callbacks
- Collections
- Jetpack Compose
- DSL (Domain Specific Language)

Inline modifier

```
fun main() {
    runSomeCode {
        println("Hello there!")
    }
}
inline fun runSomeCode(codeToRun: () -> Unit) {
    codeToRun()
}
```

Inline modifier

```
fun main() {
    println("Hello there!")
}
```

Additional resources

https://kotlinlang.org/docs/lambdas.html

https://typealias.com/start/kotlin-lambdas/

Extension functions

Extension functions

- Adding functions to types you don’t own
- Can be used on all instances of a type

Extension functions

```
fun Int.squared(): Int = this * this


fun main() {
    val number = 5
    println(number.squared())
}
```

```kotlin
suspend fun <T : Mappable<R>, R : Any> Call<T>.awaitResult(): Result<R> {
    return suspendCoroutine { continuation ->

        enqueue(object : Callback<T> {
            override fun onFailure(call: Call<T>?, error: Throwable?) {
                continuation.resume(Failure(error))
            }

            override fun onResponse(call: Call<T>?, response: Response<T>?) {
                response?.body()?.run { continuation.resume((mapToData())) }
                response?.errorBody()?.run { continuation.resume(Failure(HttpException(response))) }
            }
        })
    }
}
```

```
/**
* Instead of having a Call, adding callbacks, parsing exceptions and
similar in many places, we abstract it away.
*
* Any time we have a Call, we can just transform it to a value (Result)
directly.
*/
apiService.getWeather(cityName, API_KEY).awaitResult()
```

Additional resources

https://kotlinlang.org/docs/extensions.html#extension-functions

https://typealias.com/start/kotlin-receivers-and-extensions/

Generics

Generics

- Define behavior applicable to multiple types
- Consistent behavior, abstract types
- Classes, interfaces and functions can be generic

```kotlin
fun main() {
    val listOfNames = listOf(
        "Filip",
        "Marin",
        "Damir",
        "Luka",
        "Tomislav"
    )
}
```

```kotlin
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.size > 0) {
        elements.asList()
    } else{
        emptyList()
    }
```

```kotlin
public fun <T> listOf(vararg elements: T): List<T> =
    if (elements.size > 0) {
        elements.asList()
    } else{
        emptyList()
    }
```

Generics

- Useful for reusing logic in different cases
- Any type, the same behavior
- Can be constrained using declaration variance

Generics variance

- In - “Contravariant”, definition can safely take *in* a value of type T
- **Out -** “Covariant”, definition can safely produce *out* a value of type *T*

```kotlin
/**
* Can only produce values of type T. And not receive them.
*/
interface Producer<out T> {
   fun produce(): T
}

/**
* Can only receive values of type T. And not produce them.
*/
interface Consumer<in T> {
   operator fun consume(other: T): Int
}
```

```kotlin
/**
* Can produce and consume values of type T.
*/
interface ConsumerProducer<T> {
    fun produce(): T

    fun consume(value: T, other: T)
}
```

```kotlin
interface Producer<out T> {
    fun produce(): T
}

fun main() {
    val stringSource = object : Producer<String> {
        override fun produce() = "This is a string" //Valid
    }
}
```

```kotlin
interface Producer<out T> {
    fun produce(): T
}

fun main() {
    val stringSource = object : Producer<String> {
        override fun produce() = 505 // Error
    }
}
```

```kotlin
fun <T> countGreater(
    list: List<T>,
    threshold: T
): Int where T : CharSequence,
             T : Comparable<T> {
    return list
        .count { it > threshold }
}
```

```kotlin
fun main() {
    val threshold = "He"
    val strings = listOf("Filip", "Marin", "Ivan", "Tomislav")

    val greaterStrings = countGreater(strings, threshold)

    println(greaterStrings)
}
```

Additional resources

https://kotlinlang.org/docs/generics.html#generics.md

https://typealias.com/start/kotlin-variance/

Collections

Collections

- A collection of objects of the same type
- List, Set, Map, Array...
- Mutable & immutable collections
- Many built-in functions ready for you to use

List

- A generic ordered collection of elements

```
public interface List<out E> : Collection<E>
```

List

```
fun main() {
    val teachers = listOf("Bruno", "Luka", "Filip", "Goran")

    teachers.run {
        println(count())
        println(first())
        println(find { it.startsWith('L') })
    }
}
```

Array

- holds a fixed number of values of the same type
- If you don't have specialized low-level requirements use collections instead

```
fun main() {
    val teachers = arrayOf("Bruno", "Luka", "Filip",
"Goran")
    teachers.forEach { println(it) }
}
```

Array

```
// Boxed objects -> Integer
val integers: Array<Int> = arrayOf(1, 2, 3)


// primitive type -> int
val ints: IntArray = intArrayOf(1, 2, 3)
```

Set

- A generic unordered collection of elements
- Doesn’t support duplicates

```
public interface Set<out E> : Collection<E>
```

Set

```
fun main() {
    val teachers = setOf("Bruno", "Luka", "Filip", "Goran",
"Luka", "Filip")
    println(teachers)
}
```

Output: [Bruno, Luka, Filip, Goran]

Map

- Holds key-value pairs
- Keys are unique
- Each key can have only one value

```
public interface Map<K, out V>
```

Map

```
val nameToJob = mapOf(
    "Filip" to "Compose Dev",
    "Luka" to "Android Dev"
)

val nameJobPair: Pair<String, String> =
"Bruno" to "Professor"
```

(I)Mutability

(I)Mutability

```
fun main() {
    var myList = listOf(1, 2, 3)

    myList.add(4) // `add()` doesn't exist on the List type
}
```

(I)Mutability

```
fun main() {
    val myList = mutableListOf(1, 2, 3)
    myList.add(4)
    myList.remove(1)

    println(myList) // [2, 3, 4]
}
```

(I)Mutability

```
fun main() {
    val mySet = mutableSetOf(1, 2, 3)
    mySet.add(4)
    mySet.remove(1)

    println(mySet) // [2, 3, 4]
}
```

(I)Mutability

```
fun main() {
    val myMap = mutableMapOf(
      1 to "Luka",
      2 to "Filip",
      3 to "Bruno"
   )
   myMap.put(4, "Goran")
   myMap[5] = "David"
   myMap.remove(1)

   println(myMap) // {2=Filip, 3=Bruno, 4=Goran, 5=David}
}
```

Additional resources

https://typealias.com/start/kotlin-collections/

https://typealias.com/start/kotlin-maps/

https://kotlinlang.org/docs/collections-overview.html#collections-overview.md

Data classes

Data classes

- Used to hold data
- Compiler generates member functions for you
  - copy, toString, equals, hashCode...
- Marked with data modifier

Data classes

```
data class User(
    val name: String,
    val lastName: String,
    val job: String,
)
```

Data classes

```
fun main() {
    val filip = User("Filip", "Babic", "Compose Dev")
    val filipClone = filip.copy()

    val (name, lastName, job) = filip

    println(filip == filipClone) // True

    println(filip) // User(name=Filip, lastName=Babic, job=Compose Dev)
}
```

Data classes

```
fun main() {
    val filip = User("Filip", "Babic", "Compose Dev")
    val filipClone = filip.copy()

    val (name, lastName, job) = filip

    println(filip == filipClone) // True

    println(filip) // User(name=Filip, lastName=Babic, job=Compose Dev)
}
```

Data classes

```
fun main() {
    val filip = User("Filip", "Babic", "Compose Dev")
    val filipClone = filip.copy()

    val (name, lastName, job) = filip

    println(filip == filipClone) // True

    println(filip) // User(name=Filip, lastName=Babic, job=Compose Dev)
}
```

Data classes

```
fun main() {
    val filip = User("Filip", "Babic", "Compose Dev")
    val filipClone = filip.copy()

    val (name, lastName, job) = filip

    println(filip == filipClone) // True

    println(filip) // User(name=Filip, lastName=Babic, job=Compose Dev)
}
```

Data classes

```
fun main() {
    val filip = User("Filip", "Babic", "Compose Dev")
    val filipClone = filip.copy()

    val (name, lastName, job) = filip

    println(filip == filipClone) // True

    println(filip) // User(name=Filip, lastName=Babic, job=Compose Dev)
}
```

Additional resources

https://kotlinlang.org/docs/data-classes.html#data-classes.md

https://typealias.com/start/kotlin-data-classes-and-destructuring/

Sealed classes

Sealed classes

- Classes defined with a fixed number of types
- Can be defined only within the same file
- Can’t instantiate sealed class, only subclasses
- Useful for when pattern matching

```kotlin
sealed class HomeScreenState

object Initial : HomeScreenState()

data class Loading(val progress: Float) : HomeScreenState()

data class Ready(val data: List<String>) : HomeScreenState()

data class Error(
    val exception: Exception,
    val message: String
) : HomeScreenState()
```

```kotlin
sealed class HomeScreenState

object Initial : HomeScreenState()

data class Loading(val progress: Float) : HomeScreenState()

data class Ready(val data: List<String>) : HomeScreenState()

data class Error(
    val exception: Exception,
    val message: String
) : HomeScreenState()
```

```kotlin
fun main() {
    val state = getHomeScreenState()

    when (state) {
        is Error -> println(state.message) // Smart cast to Error
        Initial -> {
            // start an operation
        }

        is Loading -> showProgress(state.progress) // Smart cast
        is Ready -> showItems(state.data) // Smart cast
    }
}
```

```kotlin
fun main() {
    val state = getHomeScreenState()

    when (state) {
        is Loading -> showProgress(state.progress) // Smart cast
        is Ready -> showItems(state.data) // Smart cast
        // Error -> Must handle all cases or use `else` block
    }
}
```

Sealed interfaces

Sealed interfaces

- Use cases are the same as sealed classes
- Allows for greater extensibility

```kotlin
sealed interface HomeScreenState // Interface, not a class

object Initial : HomeScreenState

data class Loading(val progress: Float) : HomeScreenState

data class Ready(val data: List<String>) : HomeScreenState

data class Error(
    val exception: Exception,
    val message: String
) : HomeScreenState
```

Additional resources

https://kotlinlang.org/docs/sealed-classes.html#sealed-classes.md

https://typealias.com/start/kotlin-sealed-types/

Thank you!

Questions?