BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Mobile development and security

Session 1

Karim Karimov
Lecturer

About the course

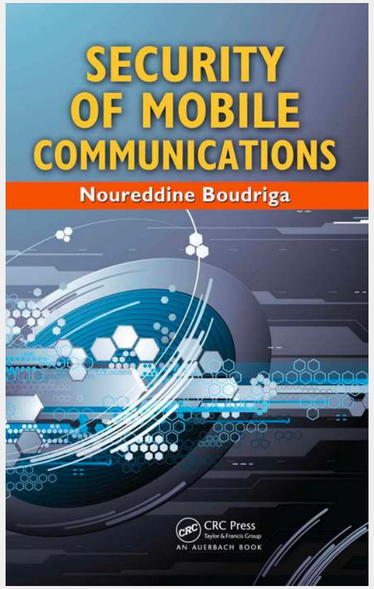
01 You will start to learn how to code in **Kotlin programming languag**e and use this knowledge in Android application development.

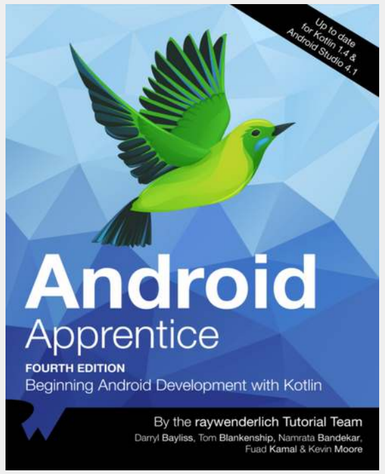
02 Through this course, you will deep dive into the internals of **Android development kit (SDK)** and learn how to use them.

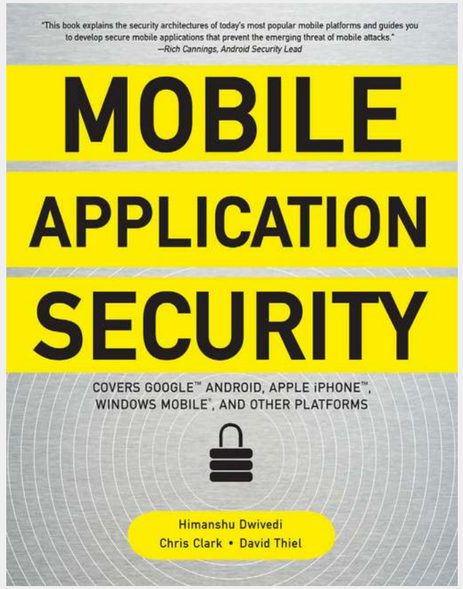
03 Theories in general **mobile device security topics** will be covered, which are applicable for all smartphones.

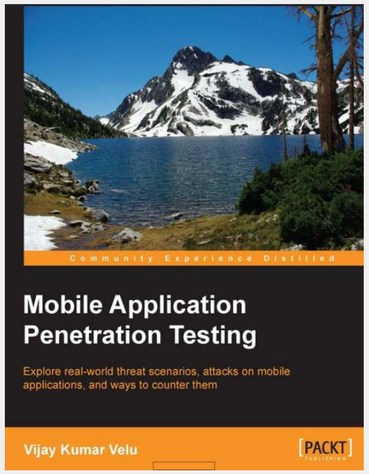
Resources


Up to date for Kotlin 1.4.x & IntelliJ IDEA 2020.3
Kotlin
Apprentice
THIRD EDITION
Beginning Programming with Kotlin
By the raywenderlich Tutorial Team
Irina Galata, Victoria Gonda, Joe Howard & Ellen Shapiro

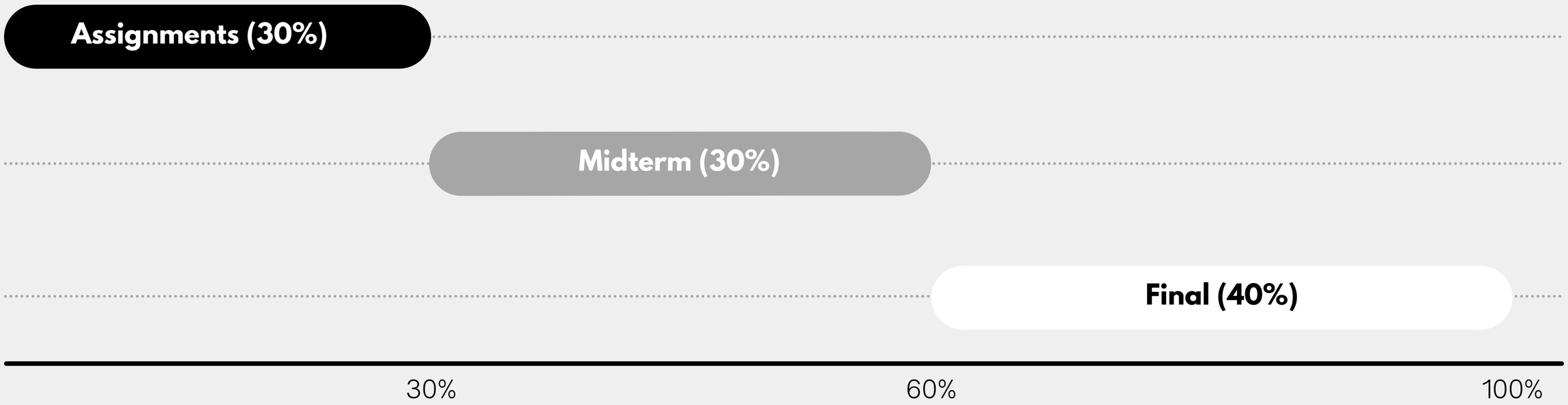

SECURITY OF MOBILE COMMUNICATIONS
Noureddine Boudriga
CRC Press
Taylor & Francis Group
AN AUERBACH BOOK


Up to date for Kotlin 1.4 & Android Studio 4.1
Android
Apprentice
FOURTH EDITION
Beginning Android Development with Kotlin
By the raywenderlich Tutorial Team
Darryl Bayliss, Tom Blankenship, Namrata Bandekar, Fuad Kamal & Kevin Moore


"This book explains the security architectures of today's most popular mobile platforms and guides you to develop secure mobile applications that prevent the emerging threat of mobile attacks."
—Rich Cannings, Android Security Lead
MOBILE
APPLICATION
SECURITY
COVERS GOOGLE™ ANDROID, APPLE iPHONE™, WINDOWS MOBILE®, AND OTHER PLATFORMS
Himanshu Dwivedi
Chris Clark • David Thiel


Community Experience Distilled
Mobile Application Penetration Testing
Explore real-world threat scenarios, attacks on mobile applications, and ways to counter them
Vijay Kumar Velu
[PACKT]

Grading



Kotlin

The Kotlin language has been around since 2011, but its popularity took off in 2017 when Google announced Kotlin's inclusion as a first-class language for Android development. In 2019, Google announced a "Kotlin-first" approach to Android development. With modern and expressive language characteristics such as those found in Apple's Swift, and 100% interoperability with Java, it's no wonder that Kotlin has been named a top 5 most-loved language by Stack Overflow users.

IDE

You can download IntelliJ IDEA from the JetBrains website at https://jetbrains.com/idea/.
There are both **Community** and **Ultimate** editions of the IDE; you'll just need the Community edition to work with basic functionalities. The Community edition is a free download, the Ultimate edition can be downloaded with your ***bhos.edu.az domain emails***.

Hello World with Kotlin

You can create **Hello.kt** file with following content to print a simple message:

```
fun main() {
  println("Hello, Kotlin!")
}
```

Variables

val keyword is used for **immutable** parameters, **var** for **mutable** ones.

```
val number: Int = 10
number = 11 // error
var variableNumber: Int = 42
variableNumber = 0 // success
```

Ranges

Closed range, which you represent like:

```
val closedRange = 0..5  // (0, 1, 2, 3, 4, 5)
```

A half-open range, which you represent like:

```
val halfOpenRange = 0 until 5 // (0, 1, 2, 3, 4)
```

A decreasing range, you can use downTo, which is inclusive:

```
val decreasingRange = 5 downTo 0 // (5, 4, 3, 2, 1, 0)
```

Loops

With ranges

```
val count = 10
var sum = 0
for (i in 1..count) {
   sum += i
}
```

With repeat

```
sum = 1
var lastSum = 0
repeat(10) {
  val temp = sum
  sum += lastSum
  lastSum = temp
}
```

With steps

```
sum = 0
for (i in 1..count step 2) {
   sum += i
}
```

Nullability

Simple use

```
var errorCode: Int?
errorCode = 100
errorCode = null
```

let() function

```
var name: String? = ""
name?.let {
    nonNull = name
}
```

elvis operator

```
var name: String? = null
val nonNull = name ?:
"default"
```

Lists

Immutable list

```
val innerPlanets = listOf("Mercury", "Venus", "Earth", "Mars")
```

Mutable lists

```
val outerPlanets = mutableListOf("Jupiter", "Saturn", "Uranus",
"Neptune", "Pluton")
outerPlanets.remove("Pluton")
```

Lambdas

Lambdas are also known as **anonymous** functions.

```
var multiplyLambda: (Int, Int) -> Int

multiplyLambda = { a: Int, b: Int -> Int
  a*b
}

val lambdaResult = multiplyLambda(4, 2) // 8
```

Lambdas

The lambda returned from this function will increment its internal counter each time it is called. Each time you call this function you get a different counter.

```
fun countingLambda(): () -> Int {
  var counter = 0
  val incrementCounter: () -> Int = {
        counter += 1
        counter }
  return incrementCounter
}
```

```
val counter1 = countingLambda()
val counter2 = countingLambda()
println(counter1()) // > 1
println(counter2()) // > 1
println(counter1()) // > 2
println(counter1()) // > 3
println(counter2()) // > 2
```

Singleton

The **singleton** pattern is one of the more straightforward of the software engineering **design patterns**. In most object-oriented languages, the pattern is used when you want to restrict a class to have a single instance during any given run of an application.
Object creates a singleton instance of a class.

```
object JsonKeys {
  const val JSON_KEY_ID = "id"
  const val JSON_KEY_FIRSTNAME = "first_name"
  const val JSON_KEY_LASTNAME = "last_name"
}
```

Lazy property

```
class Circle(var radius: Double = 0.0) {
  val pi: Double by lazy {
    ((4.0 * Math.atan(1.0 / 5.0)) - Math.atan(1.0 / 239.0)) *4.0
  }
  val circumference: Double
    get() = pi * radius * 2
}

val circle = Circle(5.0) // got a circle, pi has not been run
val circumference = circle.circumference // 31.42
// also, pi now has a value
```

lateinit

If you just want to denote that a property will not have a value when the class instance is created, then you can use the **lateinit** keyword.

```
class Lamp {
  lateinit var bulb: LightBulb
}
val lamp = Lamp()
// ... lamp has no lightbulb, need to buy some!
println(lamp.bulb)
// Error: kotlin.UninitializedPropertyAccessException:
// lateinit property bulb has not been initialized
// ... bought some new ones
lamp.bulb = LightBulb()
```

Sealed classes

Sealed classes are useful when you want to make sure that the values of a given type can only come from a particular limited set of subtypes. They allow you to define a strict hierarchy of types. The sealed classes themselves are abstract and cannot be instantiated.
Sealed classes act very much like enum classes, but also allow subtypes which can have multiple instances and have state.

Sealed classes

```
sealed class Shape {
  class Circle(val radius: Int): Shape()
  class Square(val sideLength: Int): Shape()
}
val circle1 = Shape.Circle(4)
val circle2 = Shape.Circle(2)
val square1 = Shape.Square(4)
val square2 = Shape.Square(2)
```

```
fun size(shape: Shape): Int {
  return when (shape) {
    is Shape.Circle -> shape.radius
    is Shape.Square -> shape.sideLength
  }
}
size(circle1) // radius of 4
size(square2) // sideLength of 2
```

Sealed classes vs. enum

- Sealed classes are **abstract**. This means that you can't instantiate an instance of the sealed class directly, only one of the declared subclasses.
- Related to that requirement, sealed classes can have **abstract members**, which must be implemented by all subclasses of the sealed class.
- Unlike enum classes, where each case is a single instance of the class, you can have **multiple instances** of a subclass of a sealed class.
- You can't make direct subclasses of a sealed class outside of the file where it's declared, and the **constructors of sealed classes are always private**.
- You can create indirect subclasses (such as inheriting from one of the subclasses of your sealed class) outside the file where they're declared, but because of the restrictions above, this usually doesn't end up working very well.

Generics

You've probably noticed working with List objects that you sometimes need to declare them with the type of item you expect in the list in angle brackets, such as **List<String>**, or **List<Int>**.

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

- This is a declaration of an interface, where anything conforming to this interface must be a **Collection**.
- Both **Collection** and **List** have an ***E*** in angle brackets — this is called the generic type. Since it's the same in both places, this indicates that the underlying type of the list and the collection must be the same.

Generic interfaces

A generic interface is an interface that is constrained to a generic type.

```
interface Container<T> {
  fun canAddAnotherItem(): Boolean
  fun addItem(item: T)
  fun canRemoveAnotherItem(): Boolean
  fun removeItem(): T
  fun getAnother(): Container<T>
  fun contents(): List<T>
}
```

Extension functions

Sometimes you need to extend the functionality of a specific class. And, quite often, direct inheritance is not an option — your class could already extend another class, for example, or the required class isn’t open for inheritance.

```
fun Random.randomStrength(): Int {
  return nextInt(100) + 10
}
fun Random.randomDamage(strength: Int): Int {
  return (strength * 0.1 + nextInt(10)).toInt()
}
```

End of the session