Amit Shekhar Follow
Co-Founder at Mindorks | Learning is a journey, let's learn together | Subscribe Now: https://mindork…
Nov 10, 2016 · 4 min read

Why You Must Try Kotlin For Android Development ?

Kotlin is a statically typed programming language for the JVM, Android and the browser.

Why use Kotlin for Android Development ?

- **Concise :** Drastically reduce the amount of boilerplate code you need to write.
- **Safe :** Avoid entire classes of errors such as null pointer exceptions.
- **Versatile :** Build server-side applications, Android apps or front-end code running in the browser.
- **Interoperable :** Leverage existing frameworks and libraries of the JVM with 100% Java Interoperability.

Let's see the things in more detail (Kotlin vs Java) :

- **Interoperable with Java :** When it comes to give a try to a new language, interoperability is a great thing which can help you.

Interoperable means you can reuse any Java class ever written, all Java code can work with Kotlin and vice versa. Learning Kotlin for a Java developer shouldn't be too hard. Everything you can do with Java, you can do in Kotlin. If you do not know how to do it in Kotlin, then just do it in Java and let the Kotlin plugin convert it to Kotlin. Make sure that you see what happened to your code, so that the next time you can do it yourself.

- **Null Safety :** Kotlin's type system is aimed at eliminating the danger of null references from code, also known as **The Billion Dollar Mistake**.

One of the most common pitfalls in many programming languages, including Java is that of accessing a member of a null references, resulting in null reference exceptions. In Java this would be the equivalent of a NullPointerException or NPE for short.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). For example, a regular variable of type String can't hold null:

```
var a: String = “abc”
a = null // compilation error
```

To allow nulls, you can declare a variable as nullable string, written String?:

```
var b: String? = “abc”
b = null // ok
```

Now, if you call a method or access a property on a, it's guaranteed not to cause an NPE, so you can safely say

```
val l = a.length
```

But if you want to access the same property on b, that would not be safe, and the compiler reports an error:

```
val l = b.length // error: variable ‘b’ can be null
```

But you still need to access that property, right? There are a few ways of doing that.

Checking for null in conditions :

First, you can explicitly check if b is null, and handle the two options separately:

```
val l = if (b != null) b.length else -1
```

Safe Calls :

Your second option is the safe call operator **(?.)** :

```
b?.length
```

This returns the length of b if b is not null, and null otherwise.

- **Smart Casting :**

```
// Java
if (node instanceOf Leaf) {
  return ((Leaf) node).symbol;
}
```

```
// kotlin
if (node is Leaf) {
  return node.symbol; // Smart casting, no need of casting
}
```

```
if (document is Payable && document.pay()) { // Smart
casting
  println(“Payable document ${document.title} was payed
for.”)
}
```

Kotlin uses lazy evaluation just like in Java. So if the document were not a Payable, the second part would not be evaluated in the first place. Hence, if evaluated, Kotlin knows that document is a Payable and uses a smart cast.

- **Default Arguments :** Default arguments are a feature you are missing in Java because it's just so convenient, makes your code more concise, more expressive, more maintainable and more readable.

```
class Developer(val name: String,
 val age: Int,
 val someValue: Int = 0,
 val profile: String = “”) {
}
```

```
val amit = Developer(“Amit Shekhar”, 22, 10, “Android
Developer”)
val anand = Developer(“Anand Gaurav”, 20, 11)
val ravi = Developer(“Ravi Kumar”, 26)
```

- **Named Arguments :** When it comes to readability, Named arguments make Kotlin awesome.

```
val amit = Developer(“Amit Shekhar”, age = 22, someValue =
10, profile = “Android Developer”)


// This code is not readable.
myString.transform(true, false, false, true, false)
```

```
// the below code is readable as named arguments are present
myString.transform(
 toLowerCase = true,
 toUpperCase = false,
 toCamelCase = false,
 ellipse = true,
 normalizeSpacing = false
)
```

- **Functional Programming :** While Java evolved to incorporate several functional programming concepts since Java 8, Kotlin has functional programming baked right in. This includes higher-order functions, lambda expressions, operator overloading, lazy evaluation and lots of useful methods to work with collections.

The combination of lambda expressions and the Kotlin library really makes your coding easier.

```
val numbers = arrayListOf(-42, 17, 13, -9, 12)
val nonNegative = numbers.filter { it >= 0 }
println(nonNegative)
```

```
listOf(1, 2, 3, 4) // list of 1, 2, 3, 4
.map { it * 10 } // maps to to 10, 20, 30, 40
.filter { it > 20 } // filters out 30, 40
.forEach { print(it) } // prints 30, 40
```

- **Concise Code :** When you use Kotlin instead of Java, there is a huge reduction of code in your project.

See it in the example :

```
// Java
Button button = (Button) findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
 @Override
 public void onClick(View view) {
 /* your code */
 }
});
```

```
// Kotlin
val button = findViewById(R.id.fab) as Button
button.setOnClickListener { view -> /* your code */}
```

. . .

Check out the complete guide to learn Kotlin.

A Complete Guide To Learn Kotlin For Android Development

A Complete Resources and Tutorials To Learn Kotlin For Android Development At One Place.

blog.mindorks.com

Thanks for reading this article. Be sure to click ❤ below to recommend this article if you found it helpful. It means a lot to me.

For more about programming, follow me and Mindorks , so you'll get notified when we write new posts.

Check out all the Mindorks best articles here.

Also, Let's become friends on **Twitter**, **Linkedin**, **Github** and **Facebook**.