Kotlin

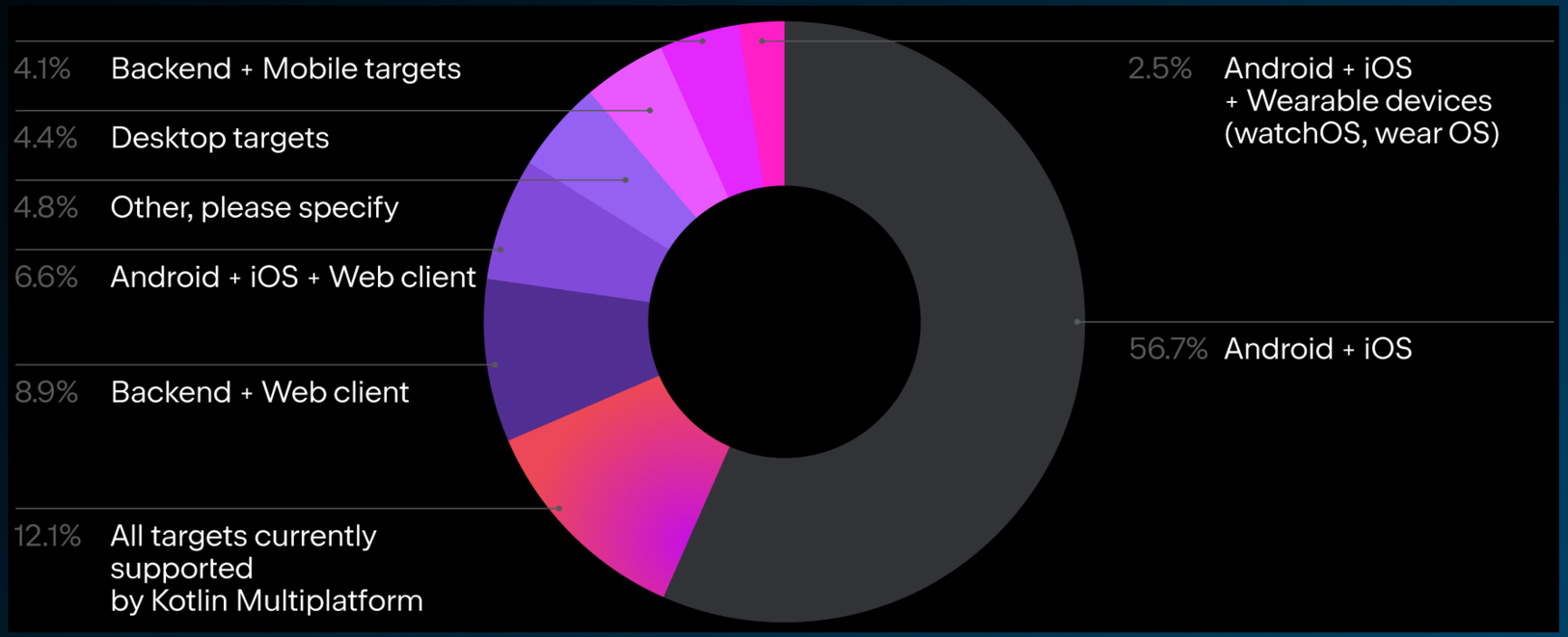
A modern programming language that makes developers happier

What is Kotlin?

- Modern - It combines all the best bits of imperative, object-oriented, and functional programming.
- Pragmatic - It combines object-oriented and functional features and is focused on interoperability, safety, clarity, and tooling support.

Features of Kotlin

- **Concise**
- **Interoperable**
- **Open source**
- **Null safety**
- **Familiar Syntax**
- **Data Classes**

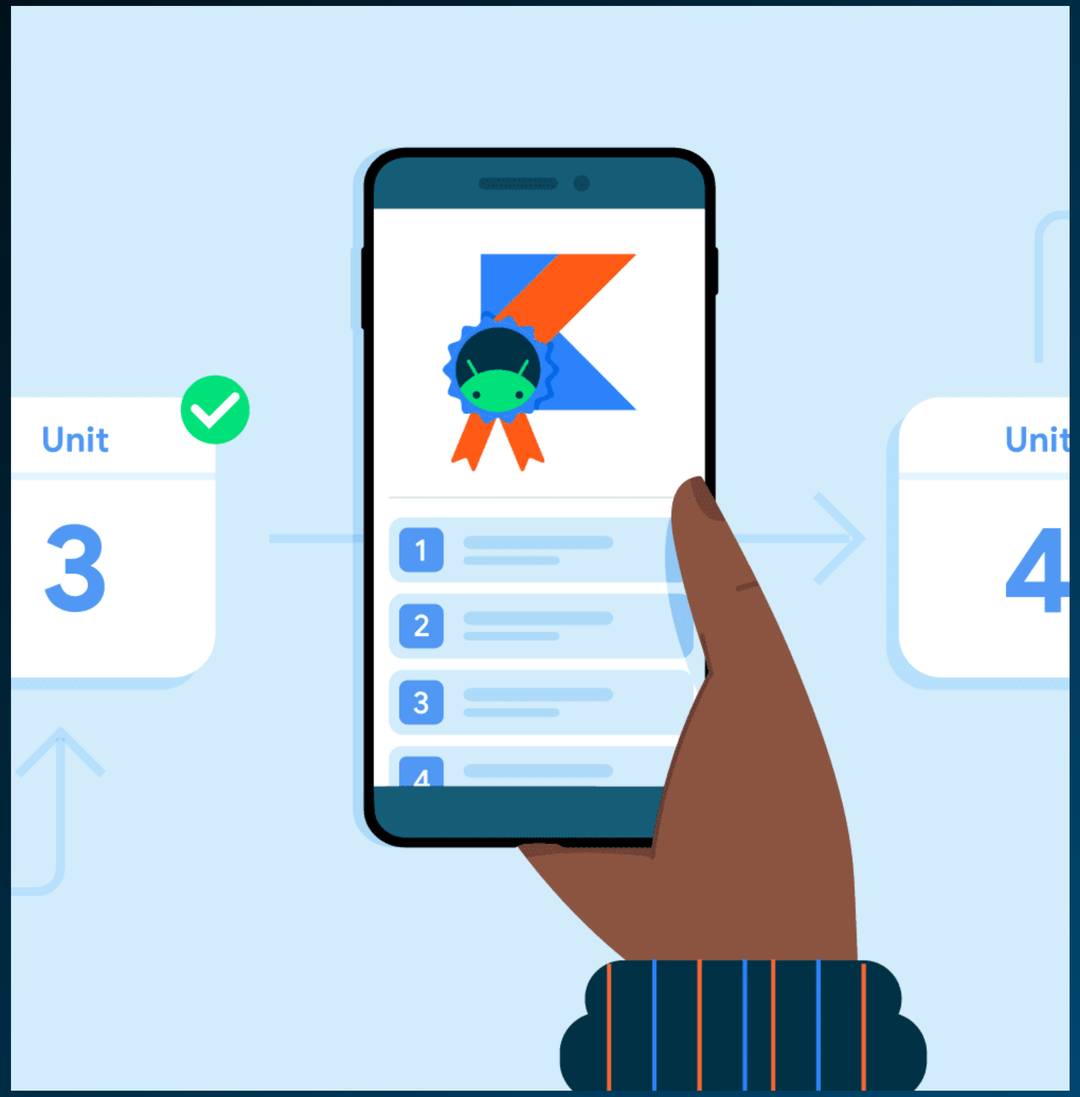


How is Kotlin better than Java?

- Coroutines
- Extension Functions
- Null Safety

Why is Kotlin primary language for android?

- Data Classes
- Co-routines
- Immutability
- Null Safety



What are Coroutines?

- Unlike many other languages with similar capabilities, async and await are not keywords in Kotlin and are not even part of its standard library. Moreover, Kotlin's concept of *suspending function* provides a safer and less error-prone abstraction for asynchronous operations than futures and promises.
- Coroutines can be thought of as light-weight threads, but there is a number of important differences that make their real-life usage very different from threads.

What are Coroutines?

JAVA:

```java
Thread thread = new Thread(new Runnable() {
@Override
public void run(){
    try {
        Thread.sleep(1000);
        System.out.println("Print this after 1 second.");
    } catch(InterruptedException e) {
        e.printStackTrace();
    }
}):
thread.start();
```

Kotlin:

```kotlin
CoroutineScope(Dispatchers.Main).launch{
  delay(1000)
  println("print this after 1 second.")
}
```

Syntax

```
package org.kotlinlang.play     //Package

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

Variables

```
fun main() {
    var name = "John"
    val birthyear = 1975

    println(name)
    println(birthyear)
}
```

The difference between *var* and *val* is that variables declared with the *var* keyword can be changed/modified, while *val* variables cannot.

Variables

```
fun main() {
    var name: String
    name = "John"
    println(name)
}
```

You can also declare a variable without assigning the value, and assign the value later. However, this is only possible when you specify the type.

String

```
fun main() {
    var txt = "Hello World"
    println(txt[0]) // first element (H)
    println(txt[2]) // third element (l)
}
```

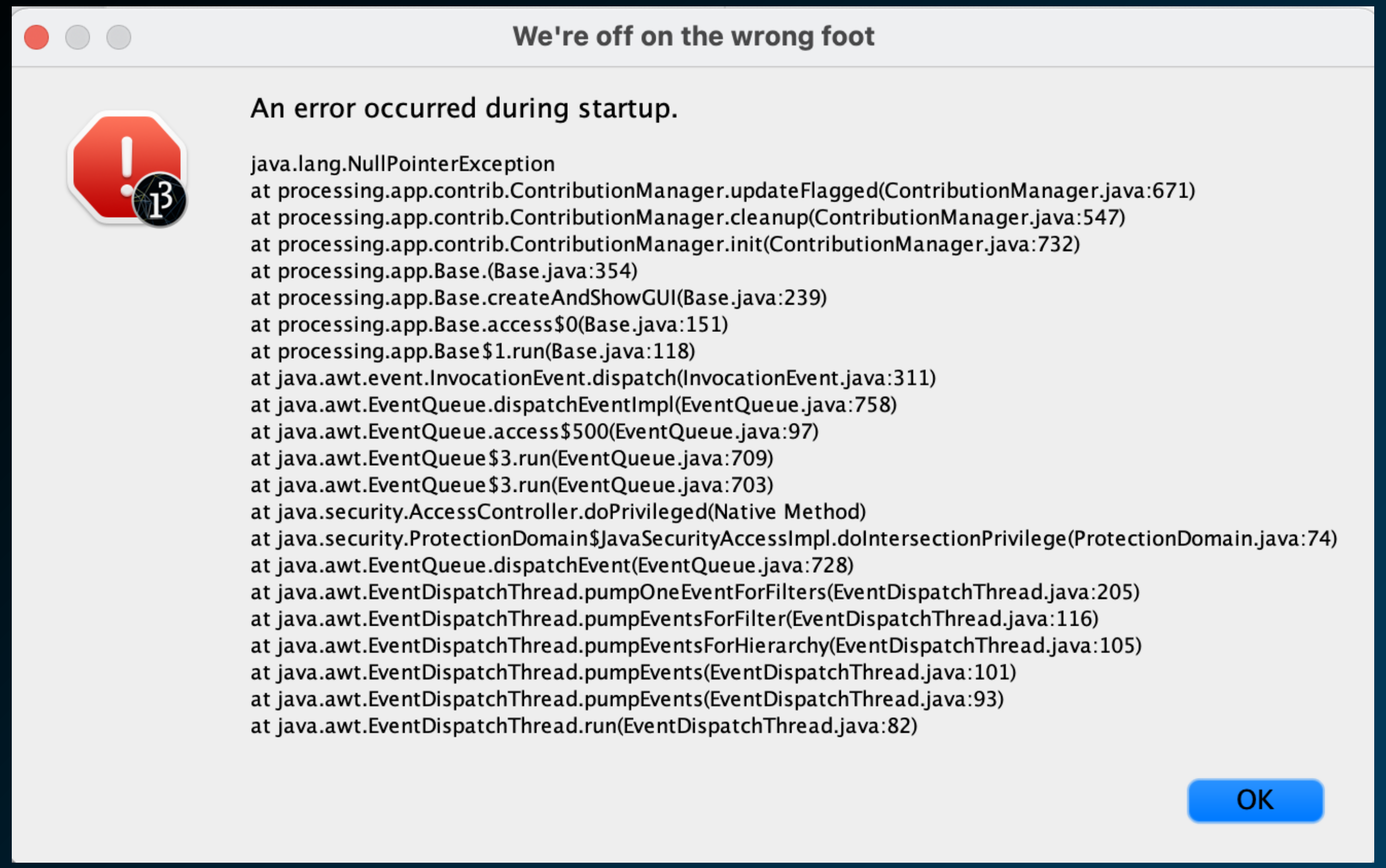
To access the characters (elements) of a string, you must refer to the index number inside square brackets.

String

```kotlin
fun main() {
    var txt = "Hello World"
    println("The length of the txt string is: " + txt.length)
}
```

A String in Kotlin is an object, which contain properties and functions that can perform certain operations on strings, by writing a dot character (.) after the specific string variable. For example, the length of a string can be found with the length property

Billion Dollar Mistake



Null Safety

```
fun main() {
    var nullable: String? = "You can keep a null here"
    nullable = null

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

It’s a modern approach to make Null Pointer Exception Compile-time error, not Run-time error

Classes

```
// without any properties
class Customer

// class with two properties
class Contact(val id: Int, var email: String)

fun main() {
    // creating instance of classes
    val customer = Customer()
    val contact = Contact(1, "john@gmail.com")

    println(contact.id)
    contact.email = "jane@gmail.com"
}
```

Inheritance

```
open class Dog {
    open fun sayHello() {
        println("wow wow!")
    }
}

class Yorkshire : Dog() {
    override fun sayHello() {
        println("wif wif!")
    }
}

fun main() {
    val dog: Dog = Yorkshire()
    dog.sayHello()
}
```

Control Flow

When

```kotlin
fun main() {
    cases("Hello")
    cases(1)
    cases(0L)
    cases("hello")
}

fun cases(obj: Any) {
    when (obj) {
        1 -> println("One")
        "Hello" -> println("Greeting")
        is Long -> println("Long")
        else -> println("Unknown")
    }
}
```

Instead of the widely used ***switch*** statement, Kotlin provides a more flexible and clear '***when***' construction. It can be used either as a statement or as an expression.

For-loop

```
fun main() {
    val cakes = listOf("carrot", "cheese", "chocolate")

    for(x in cakes){
        println(x)
    }
}
```

for in Kotlin works the same way as in most languages.

While-loop

```
fun main() {
    while (condition) {
        // code block to be executed
    }
}
```

while construct works similarly to most languages as well

Conditionals

```
fun main() {
    if (condition) {
        // block of code to be executed if the condition is true
    } else {
        // block of code to be executed if the condition is false
    }
}
```

Ternary Operator

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

    println(max(99, -42))
}
```

There is no ternary operator condition ***?*** then ***:*** else in Kotlin. Instead, ***if*** may be used as an expression

Ranges

```
fun main() {
    for (nums in 5..15) {
        println(nums)
    }
}
```

With the ***for*** loop, you can also create ranges of values with "***..***"

Special Classes

Data classes

```
data class Person(val name: String)

fun main() {
    val person1 = Person("John")
    val person2 = Person("John")
    println("person1 == person2: ${person1 == person2}")
}
```

Data Classes make it easy to create classes that are used to store values.

Enum

```
enum class State {
    IDLE, RUNNING, FINISHED
}

fun main() {
    val state = State.RUNNING
    val message = when (state) {
        State.IDLE -> "It's idle"
        State.RUNNING -> "It's running"
        State.FINISHED -> "It's finished"
    }
    println(message)
}
```

Enum classes are used to model types that represent a finite set of distinct values, such as directions, states, modes, and so forth.

Collections

List

```
fun main() {
    var list = mutableListOf(1,2,3,4)
    list.add(5)
    println(list)
}
```

List is an ordered collection with access to elements by indices

Set

```
fun main() {
    var st = setOf(1,2,2,3)
    println(st)
}
```

Set is a collection of objects without repetitions

Map

```
fun main() {
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
    println("All keys: ${numbersMap.keys}")
    println("All values: ${numbersMap.values}")
}
```

Map (or dictionary) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value. The values can be duplicates.

Filter

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    val longerThan3 = numbers.filter { it.length > 3 }
    println(longerThan3)
}
```

filter function enables you to filter collections.

Scope Functions

Let

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four", "five")
    numbers.filter { it.length > 3 }.let {
        println(it)
        // and more function calls if needed
    }
}
```

let can be used to invoke one or more functions on results of call chains.

With

```
fun main() {
    val numbers = mutableListOf("one", "two", "three")
    with(numbers) {
        println("'with' is called with argument $this")
        println("It contains $size elements")
        println("First element: ${first()}")
        println("Last element: ${last()}")
    }
}
```

with is a non-extension function that can access members of its argument concisely.

Apply

```
data class Person(var name: String, var age: Int = 0, var city: String = "")

fun main() {
    val adam = Person("Adam").apply {
        age = 32
        city = "London"
    }
    println(adam)
}
```

apply executes a block of code on an object and returns the object itself. Inside the block, the object is referenced by *this*. This function is handy for initializing objects.

About Me

LinkedIn:
https://www.linkedin.com/in/vandit-vasa-bb506317a/

ISTE.VIT

ISTE_VIT_VELLORE

INDIAN SOCIETY FOR TECHNICAL EDUCATION

ISTE-VIT