

An introduction to Kotlin for Java Developers

Antonis Lilis, Mobile Engineer @ Automattic

Some History

- **2010:** JetBrains starts developing Kotlin
- **2011:** JetBrains **unveiled** Project Kotlin, a new language for the JVM
- **2012:** JetBrains **open sourced** the project under the Apache 2 license
- **2016:** Kotlin **v1.0** is released

- **2017:** Kotlin support in **Spring** Framework
- **2017:** Google announced **first-class support for Kotlin on Android**

- Kotlin is **technically 10**, but in reality **4 years old**

The Kotlin Language

- **Statically Typed**
 - Type validation at compile time
- Supports **Type Inference**
 - Type automatically determined from the context
- Both **Object Oriented** and **Functional**
- **First-class functions**
 - You can store them in variables, pass them as parameters, or return them from other functions
- Was designed with **Java Interoperability** in mind

Constants and Variables

- **val** (from value)
 - **Immutable** reference
- **var** (from variable)
 - **Mutable** reference
- **Nullable** Types
 - Defined **Explicitly**

```
val someInt: Int = 42
var someString = "forty-two"
var someValue: Int? = 23
```

No
semicolon
here ;)

```
someInt = 23 //It is constant
someString = "twenty-three"
someString = 5 //It is a String
someString = null //Cannot be null
someValue = null
```

Control Flow

- Classic loops:
 - **if**
 - **for**
 - **while / do-while**
- **when**
 - Replaces the switch operator
 - No breaks, no errors

```
when (x) {  
    1 -> print("x == 1")  
    2 -> print("x == 2")  
    else -> { //block  
        print("not 1 or 2")  
    }  
}
```

Functions

- **Named** arguments
- Can be declared at the **top level** of a file (without belonging to a class)
- Can be **Nested**
- Can have a **block or expression body**

```
fun max(a: Int, b: Int): Int { //name – parameters – return type  
    return if(a>b) a else b //function block body  
}
```

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

```
max(a = 1, b = 2) //call with named arguments
```

```
max( a: 1, b: 2)
```

Functions

- **Default** parameter values
 - No method overloading and boilerplate code

```
fun doSomethingWith(letter: Char, number: Int = 42) {  
    val res = "The letter is ${letter} and the number is $number"  
    println(res)  
}
```

```
doSomethingWith(letter = 'C', number = 1)
```

```
doSomethingWith(letter = 'A')
```

```
doSomethingWith( letter: 'A')
```

Simple string
Interpolation

Classes

```
class MyView : View {  
    constructor(ctx: Context): super(ctx) {  
        //Initialization stuff  
    }  
    //...  
}
```

```
class MyViewShort(ctx: Context) : View(ctx) {  
    //...  
}
```

```
class Car(val brand: String, val isUsed: Boolean = false)  
  
val car = Car(brand: "Ford")
```

data classes:
autogenerated
implementations of
universal **methods**
(equals, hashCode
etc)

```
data class Bike(val brand: String, val isUsed: Boolean = false)
```


Modifiers

- **Access** modifiers
 - **final** (default)
 - **open**
 - **abstract**
- **Visibility** modifiers
 - **public** (default)
 - **internal**
 - **protected**
 - **private**

***"Design and document for inheritance
or else prohibit it"***

Joshua J. Bloch, Effective Java

*ps. Lukas Lechner has written a series of articles on
"How Effective Java influenced Kotlin" (<http://lukle.at>)*

Properties

- **first-class** language feature
- combination of the **field**
and its **accessors**

```
class House {  
  
    var street: String = "Ermou"  
    var number: String = "1"  
    var city: String = "Athens"  
  
    var state: String? = null  
  
    var zip: String = ""  
    set(value) {  
        state = "TK.$value"  
    }  
  
    val prettyAddress: String  
    get() = "$street $number, $city"  
  
}
```

No static keyword

- **Top-level** functions and properties
(e.g. for utility classes)
- **Companion objects**
- The **object** keyword:
declaring a class and creating an instance
combined (**Singleton**)

```
class Foo {  
    companion object {  
        fun bar() {  
            //...  
        }  
    }  
}
```

```
object Singleton {  
    fun doSomething() {  
        //..  
    }  
}
```

Foo.bar()

Singleton.doSomething()

Extensions

- Enable **adding methods and properties** to other people's classes
 - Of Course without access to private or protected members of the class

```
fun String.lastChar(): Char  
    = this.get(this.length - 1)
```

```
val String.lastChar: Char  
    get() = get(length - 1)
```

```
val last: Char = "hi".lastChar()
```

```
"hello".lastChar
```

Null Checks

- Safe-call operator ?.
- The **let** function
- Elvis operator ?:



```
fun strLen(s: String?): Int? = s?.length
```

```
fun strLen(s: String?): Int = s?.length ?: 0
```

```
fun sendEmailTo(email: String) { }
```

```
var email: String? = "yole@example.com"  
email?.let { sendEmailTo(it) }
```

```
email = null  
email?.let { sendEmailTo(it) } //won't be executed
```

***"I call it my billion-dollar mistake.
It was the invention of the null reference in 1965"***

Tony Hoare

Not-null assertion operator !!

```
fun rootOfAllEvils(s: String?) {  
    val sNotNull: String = s!!  
    println(sNotNull.length)  
}
```

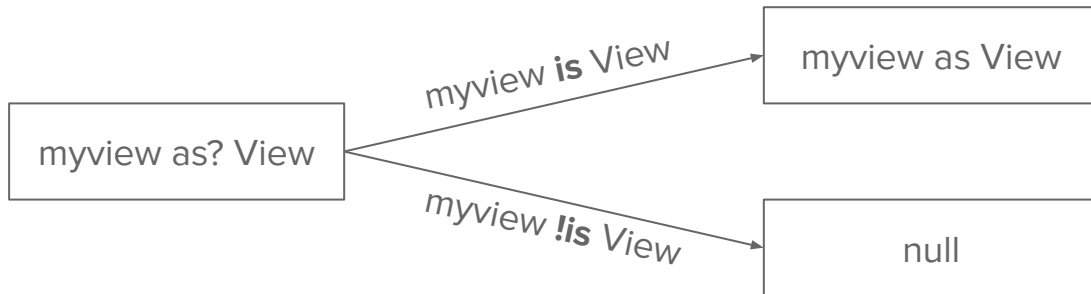


Safe Casting

- Safe cast operator **as?**
- Smart cast
 - combining type checks and casts

```
var myview: MyView? = MyView(ctx)
val view = myview as? View

if (view is MyView) {
    view.bar()
}
```



Lamdas and Higher Order Functions

```
val sum = { x: Int, y: Int -> x + y }
```

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

```
println(sum(1, 2))
```

```
fun twoAndThree(operation: (Int, Int) -> Int) {  
    val result = operation(2, 3)  
    println("The result is $result")  
}
```

```
twoAndThree(operation = {a, b -> a + b})  
twoAndThree { a, b -> a * b }
```


Collections

- Kotlin **enhances** the **Java** collection classes (**List**, **Set**, **Map**)

```
class Car(val brand: String, val age: Int, val horsepower: Int)
```

```
val fleet = listOf(  
    Car( brand: "Ford",   age: 1,   horsepower: 100),  
    Car( brand: "Mazda",  age: 2,   horsepower: 120),  
    Car( brand: "Opel",   age: 2,   horsepower: 95))
```

```
fleet.maxBy { it.horsePower }
```

```
fleet.filter { it.age == 2 }
```

```
fleet.filter { it.age == 2 }.maxBy { it.horsePower }
```

```
fleet.forEach { print("brand: $it.brand") }
```

Chained
Calls

Delegation

- **Composition over Inheritance** design pattern
- **Native support** for delegation (implicit delegation)
- **Zero Boilerplate** code
- Supports both **Class Delegation** and **Delegated Properties**

Class Car **inherits from an interface** Nameable and **delegates** all of its public **methods to a** delegate **object** defined with the **by keyword**

```
interface Nameable {  
    var name: String  
}  
  
class Ford : Nameable {  
    override var name = "Ford"  
}  
  
class Car(name: Nameable)  
    : Nameable by name  
  
val car = Car(Ford())  
print(car.name) //Ford
```

Domain-specific language construction

- Kotlin provides mechanisms for creating internal DSLs that use exactly the **same syntax** as all language features and are fully **statically typed**

```
object start

infix fun String.should(x: start)
    = StartWrapper( value: this)

class StartWrapper(val value: String) {
    infix fun with(prefix: String)
        = value.startsWith(prefix)
}

"kotlin".should(start).with( prefix: "kot")

"kotlin" should start with "kot"
```

```
table { this: TABLE
    tr { this: TR
        td { this: TD
            + "Cell A"
        }
        td { this: TD
            + "Cell B"
        }
    }
}
```



Hello World

- Kotlin files have **.kt extension**
- You can also try your code in **REPL** (Read-Eval-Print-Loop)
- Online play.kotlinlang.org

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

```
Hello, world!!!
```

Let's Mix with some Java

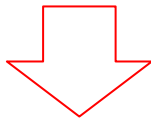
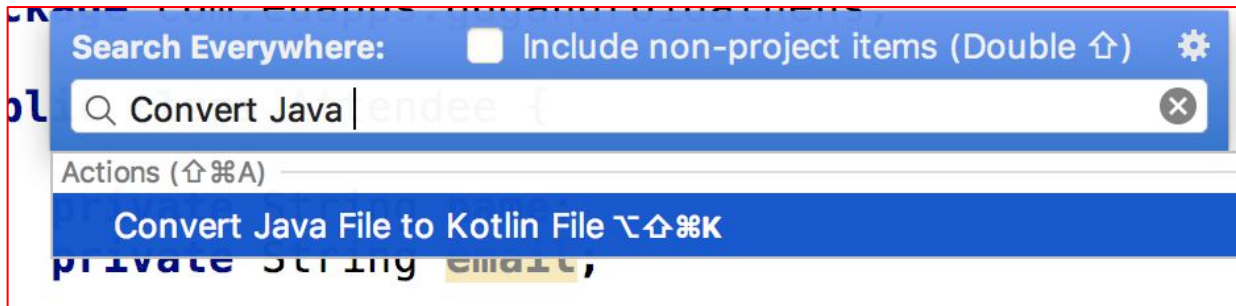
```
public class Attendee {  
    private String email;  
    private String name;  
  
    public Attendee(String email, String name) {  
        this.email = email;  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

Java from Kotlin

- You can call Java code from you Kotlin files transparently

```
fun main(args: Array<String>) {  
    val attendee = Attendee( email: "antonis.lilis@gmail.com", name: "Antonis")  
    println("Hello ${attendee.name}")  
}
```

Convert Java to Kotlin



```
2  
3 class Attendee(private val email: String, val name: String)  
4
```


Utility

- **Top-level** computed **property**
- String **extension**

```
val String.isValidEmail: Boolean
    get() {
        //Dummy implementation
        return isEmpty() &&
            contains('@') &&
            contains('.')
    }
```

Kotlin from Java

- In most cases the integration is **Seamless**
- Some **Kotlin features do not exist in Java** (e.g. top level functions or properties)
- In such cases **conventions** are used
- In our case a **static class is generated** for the **top-level** declarations

```
public class Conference {  
    private String title;  
    private List<Attendee> attendees;  
  
    public Conference(String title) {  
        this.title = title;  
        this.attendees = new ArrayList<Attendee>();  
    }  
  
    public void addAttendee(String name, String email) {  
        if (UtilsKt.isValidEmail(email)) {  
            Attendee attendee = new Attendee(name, email);  
            attendees.add(attendee);  
        }  
    }  
}
```

Libraries & Resources

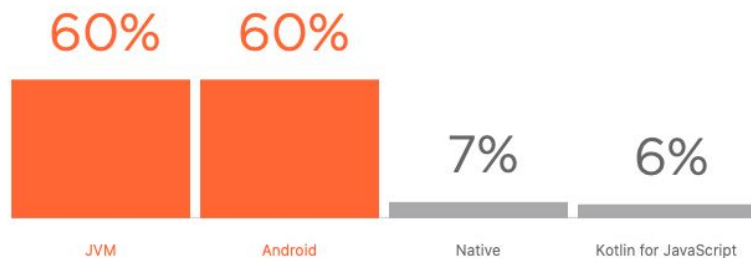
- The Kotlin **standard library** is great
- You can use **Any Java Library** since Java and Kotlin are 100% interoperable
- **Kotlin libraries:** a nice curated list at <https://kotlin.link>
- Kotlin **popularity is growing** and resources become more abundant

CHANGE IN PROGRAMMING LANGUAGE USE, 2018-2019		
01	Dart	532%
02	Rust	235%
03	HCL	213%
04	Kotlin	182%

[REF: [GitHub](#)]

Kotlin also lives outside the JVM

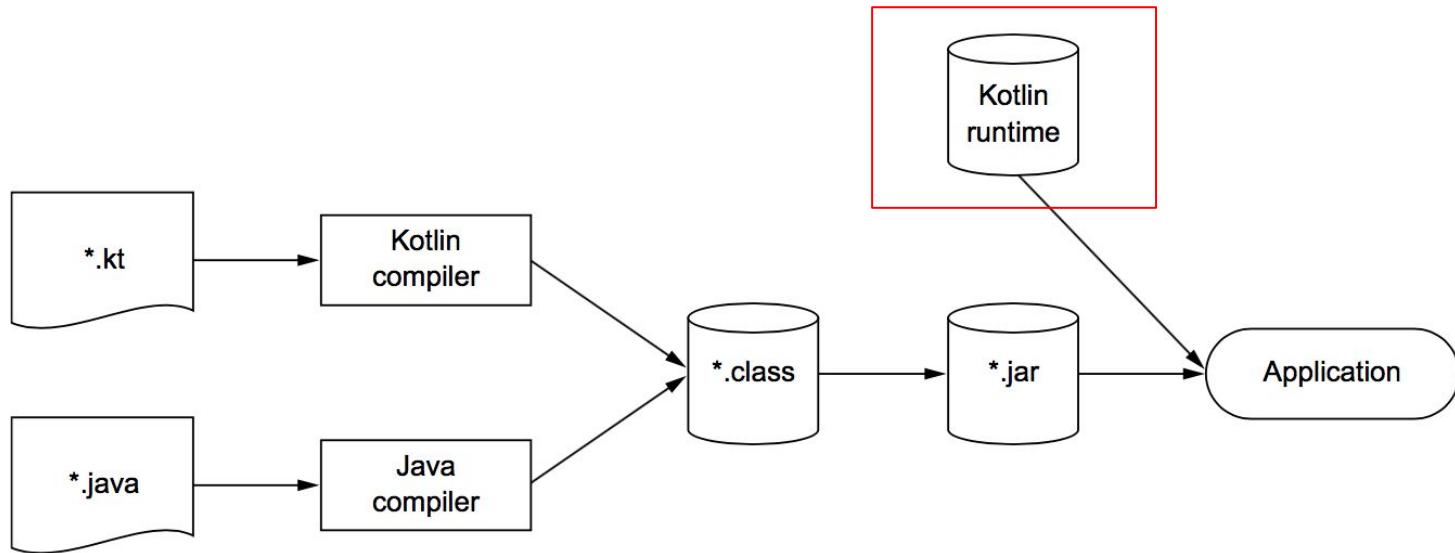
- Kotlin 1.1 (March 2017): officially released the **JavaScript** target, allowing you to compile Kotlin code to JS and to run it in your browser
- Kotlin/**Native** v0.6 (Valentine's Day 2018): Better support for native targets (e.g. **iOS**, **WebAssembly**, **Windows**)
- Kotlin 1.4 (August 2020)
Kotlin **Multiplatform** Mobile Alpha



[REF: [Jetbrains](#)]

Any Disadvantages?

- An app built with Kotlin will likely result in a **larger file** package size than one built purely in Java (and a bigger method count)
- The **build time** for Kotlin is sometimes a little slower



Name shadowing

- Probably a design flaw
- Produces an warning in IDEA

```
fun xShadow(x : Int) {  
    val x = 1 //Java: already defined error  
    if (x == 0) {  
        val x = 2 //Java: already defined error  
    }  
    println ("x = $x")  
}  
xShadow(3)  
x = 1
```

Null Safety vs Java Interoperability

```
public class Utils {  
    public static String format(String text) {  
        return text.isEmpty() ? null : text;  
    }  
}
```



```
@Test fun testFormat1() {  
    //Platform type String! (String or String?)  
    val f = Utils.format( text: "" )  
    assertNull(f)  
}  
  
@Test fun testFormat2() {  
    val f: String = Utils.format( text: "" ) ?: ""  
    assertNotNull(f)  
}  
  
@Test fun testFormat3() {  
    val f: String? = Utils.format( text: "" )  
    assertNull(f)  
}  
  
@Test fun testFormat4() {  
    val f: String = Utils.format( text: "" )!!  
    assertNotNull(f)  
}
```



In my 

My World

Objective-C

```
- (NSString*) joinAString:(NSString*) stringParam andAnInteger:(int) intParam {  
    return [NSString stringWithFormat:@"%s%d", stringParam, intParam];  
}  
  
[self joinAString:@"Voxxed" andAnInteger:2018];
```

Swift

```
func join(aString: String, anInteger: Int) -> String {  
    return "\(aString)\(anInteger)"  
}  
  
self.join(aString: "Voxxed", anInteger: 2018)
```

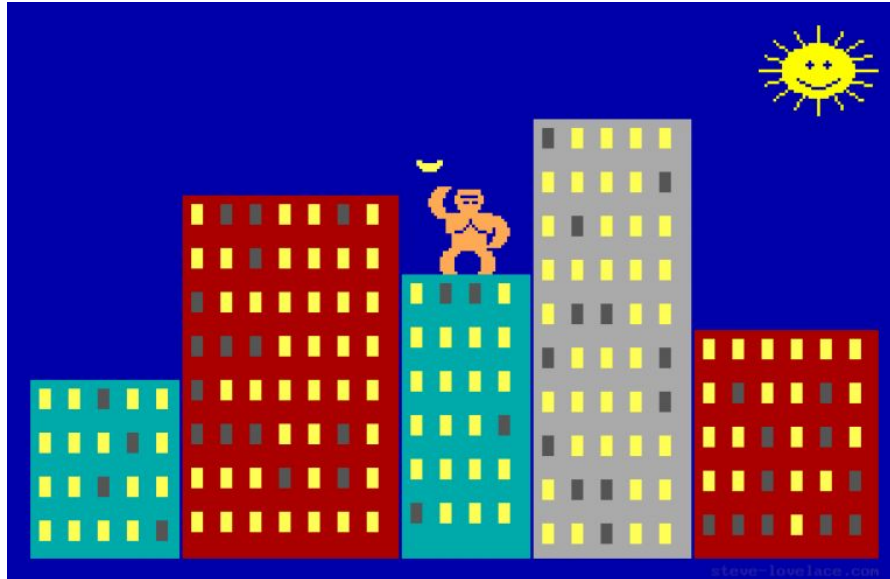
Kotlin

```
fun join(aString: String, anInteger: Int): String {  
    return "$aString$anInteger"  
}  
  
this.join(aString = "Voxxed", anInteger = 2018)
```

}
Similar
Syntax

Met Java in 2000

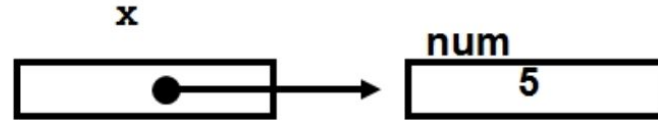
- Freshman in the university
- The only language I knew was Basic (not the Visual one)



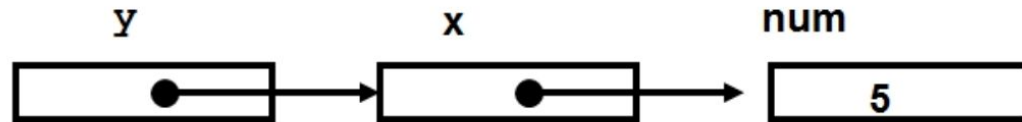
Java was Cool

- Java was new and exciting
- Java was creating new developer communities
- Developers coming from C++ and other languages were happy

```
int num = 5;  
int *x = &num;
```

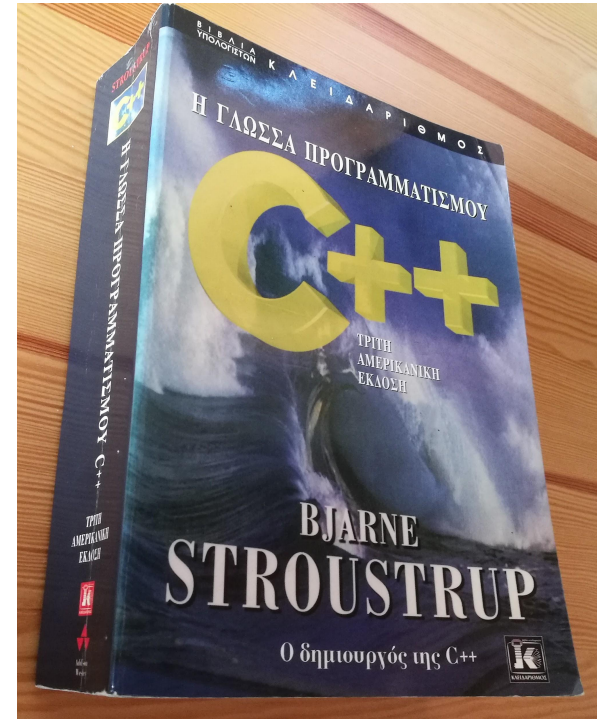


```
int **y = &x;
```



Hardcore Devs

- C++ Devs talked about
 - the superior performance
 - How powerful C++ is
- Java Developers
 - They did not care
 - The code was easier to write
 - Maintainable
 - Safer: No manual memory management
 - Happy developers



Two decades later

- I feel that kotlin is like Java
- It is new, cool and it solves problems



Why Kotlin?

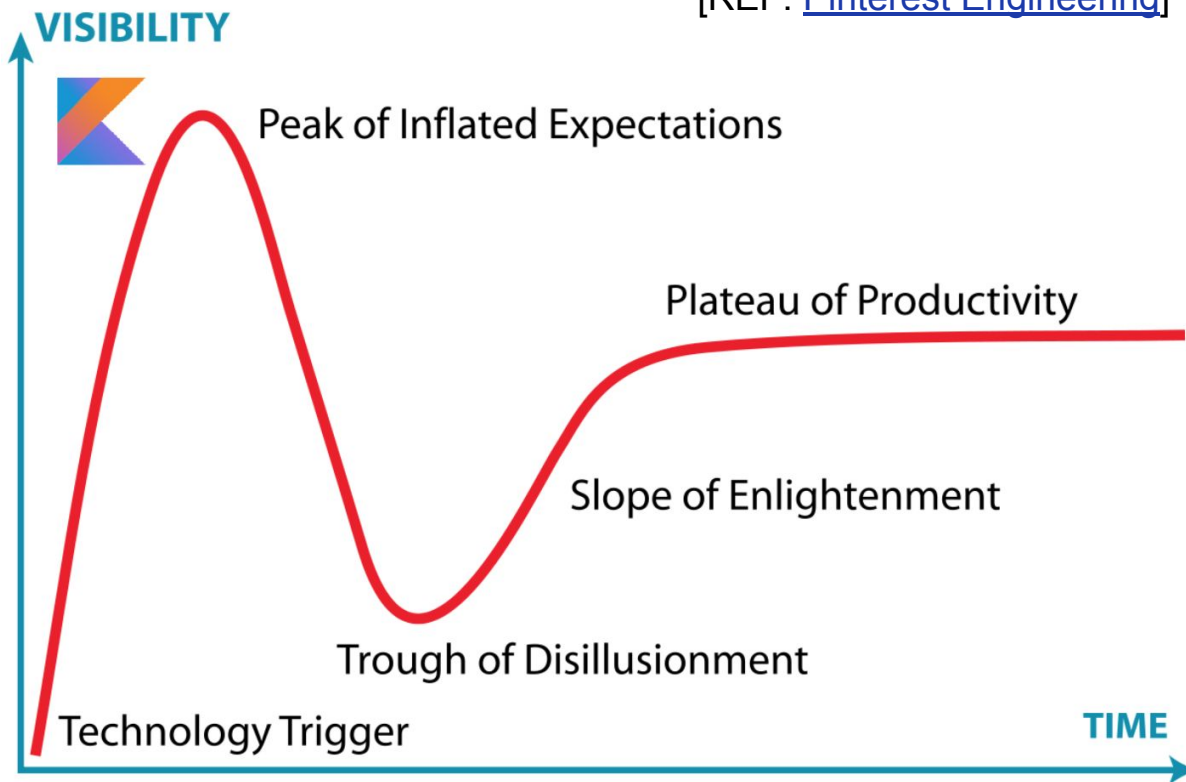
1. Makes **writing code** easier
2. Has incremental **learning curve**
3. Has nice **features**
4. Follows modern **programming language trends**
5. Can be easily **mixed with Java**
6. Suitable for **incremental adoption**
7. On **Android** there is no way back
8. Is designed by **Jetbrains** that makes some of the best developer tools
9. Is **supported** by major vendors (**Google**, Spring etc)
10. Kotlin has a **growing community**

Final Thoughts

- **Not so popular yet**
 - **34th** in [TIOBE Index](#) for May
 - Competed with C for language of the year 2017
 - Growing fast in [PYPL Popularity Index](#) (**12th**)
 - [GitHub](#) Language Growth for 2019 (**4th**)
- Development **Stability**
- **Reversibility**

Kotlin is here to stay

[REF: [Pinterest Engineering](#)]



Thank you!

Questions?