Hello, Kotlin!

Zoltán Domahidi

10 YEARS

2 YEARS

✉ domahidizoltan@gmail.com

in /in/domahidizoltan

/domahidizoltan

- Meet Kotlin

- Learn basic Kotlin code
  (variables, functions, conditions, collections, loops, classes)

- Demo

About Java

- Industry standard programming language
- Runs on a wide range of platforms
- Great performance
- Great ecosystem and community
- Great virtual machine (JVM)

… but it has some pain points

- Very old and keeps backward compatibility
- Evolves slowly
- Verbose

Some programming languages were born
and tried to give better alternatives
while keeping the benefits of the Java ecosystem

About Kotlin

- Created by JetBrains
- Multi-paradigm modern programming language running on JVM
- Fully interoperable with Java
- Android, server-side, frontend, multiplatform, native
- Great ecosystem and community
- Easy to learn expressive syntax (makes you more productive):
    - Safer code design and fewer errors (i.e.: null safety, immutability)
    - Coroutines
    - Easy to write Domain Specific Languages
- Officially supported by Google and Spring
- Used for research by many universities
- Used in production by many companies and organisations

Stanford University

Imperial College London

Johns Hopkins University

Penn

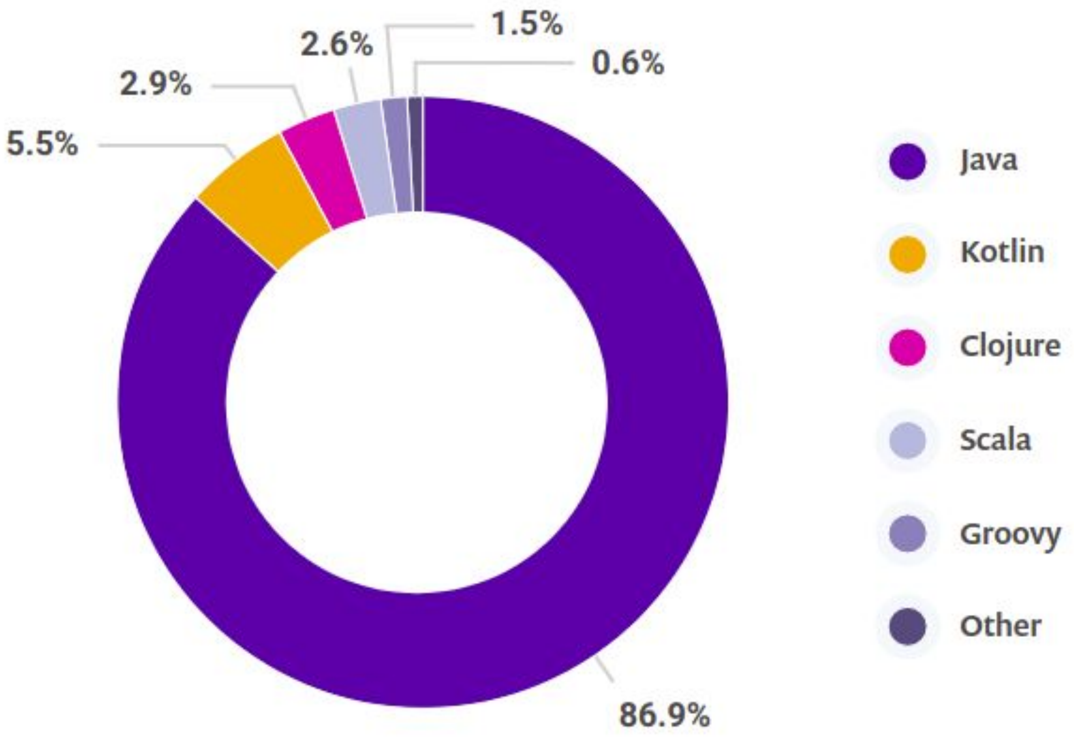
UCLA

Duke

Uber

coursera

Snyk JVM Ecosystem Report 2020



Let’s have some *fun*!

Hello, World!

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

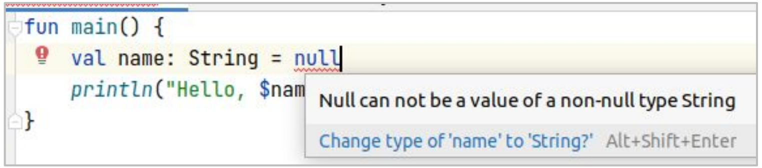
- **`fun`** - keyword for function declaration (**`public`** by default)
- **`main`** - main function to start application (no args required, no **`static`**)
- **`println`** - prints the given string to standard output (no **`System.out`**)
- no class required (with same filename)
- semicolons are optional

```java
public class HelloKotlinDemo {
   public static void main(String[] args) {
       System.out.println("Hello, World!");
   }
}
```

Hello, World! (variables)

```
fun main() {
    val name = "Kotlin"
    println("Hello, $name!")
}
```

- **val** - final/immutable variable declaration (use **var** for mutability)
- string-interpolation / templating (no concatenation; expression also allowed)
- type declaration is optional (type-inference, use **val name:String** for explicit type)
- you must be explicit if you want to use **null** value (**val name: String? = null**)



```
fun main() {
    val name: String = null
    println("Hello, $nam
}
```

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

Change type of 'name' to 'String?' Alt+Shift+Enter

```
public class HelloKotlinDemo {
    public static void main(String[] args) {
        String name = "Java";
        System.out.println("Hello, " + name + "!");
    }
}
```

Hello, World! (functions)

```
fun main() {
   println(hello("Kotlin"))
}
private fun hello(name: String): String {
   return "Hello, $name!"
}
```

- everything is **public** by default (you must be explicit with **private**)
- function arguments require type
- functions require return type ...
- ... unless you can write it with a single statement (return not needed + type inference)

```
private fun hello(name: String) = "Hello, $name!"
```

```
public static void main(String[] args) {
   System.out.println(hello("Java"));
}
private static String hello(String name) {
   return "Hello, " + name + "!";
}
```

Hello, World! (functions)

```
hello("Kotlin", 2) //Hello, KotlinKotlin!
private fun hello(name: String = "World", times: Int = 3) =
    "Hello, ${name.repeat(times)}!"
```

- everything is an object
- complex expressions could be written within string templates by using **${ }**
- function arguments can have default values to use in case they are not referenced

```
hello("Kotlin") //Hello, KotlinKotlinKotlin!
hello() //Hello, WorldWorldWorld!
```

- function arguments can be referenced by name

```
hello(name = "Kotlin", times = 2) //Hello, KotlinKotlin!
hello(times = 2, name = "Kotlin") //Hello, KotlinKotlin!
hello(times = 2) //Hello, WorldWorld!
```

- no more method overloading

```
hello("Java", 2);
private static String hello(String name, int times) {
    String longName = name.repeat(times);
    return "Hello, " + longName + "!";
}
```

Conditional expression

```
val number = 0
val sign =
   if (number < 0) -1
   else if (number == 0) 0
   else 1
```

- same condition syntax as in Java (condition, statement, else branch)
- **`if`** statement is an expression in Kotlin (it’s result could be assigned to a variable)
- last statement of each block is an implicit return statement

```
int number = 1;
int sign = 0;
if (number < 0) sign = -1;
else if (number == 0) sign = 0;
else sign = 1;
```

When expression

```kotlin
val number: Any = 0
val sign = when (number) {
    0 -> 0
    -1, -2 -> -1
    "-3", "-4" -> -1
    is String -> 0
    in 1..10 -> 1
    else -> 0
}
```

- **`when`** is similar to **`switch`** in Java, but much smarter (pattern matching)
- it is also an expression (returns a value)
- compare single value
- compare multiple values
- check value within range
- compare enums or check value is within enum range
- type comparison
- compare any type of object (even multiple type comparison within same **`when`** statement)
- **`else`** - anything not defined

Collections (lists)

```
val list = listOf("one", "two", "three") //[one, two, three]
list + "four" //[one, two, three, four]
list - "two" //[one, three]
list[2] = "3" //error
```

- create collections by using helper methods
- type aware operators to add and remove items
- access items by index (array-like)
- every operation on collections creates a copy of the original variable
- collections are immutable by reference (use **`mutable*`** versions if needed)

```
List<String> list = new ArrayList<>();
list.addAll(Arrays.asList("one", "two", "three")); //[one, two, three]
List<String> copy = new ArrayList<>(list);
<copy>.add("four"); //[one, two, three, four]
<copy>.remove("two"); //[one, three]
<copy>.set(2, "3"); //[one, two, 3]
```

Collections (sets)

```kotlin
val set = setOf("one", "two") //[one, two]
set union setOf("two", "three") //[one, two, three]
set intersect setOf("two", "three") //[two]
set subtract setOf("two", "three") //[one]
```

- built in **`union`**, **`intersect`** and **`subtract`** methods for clarity

```java
Set<String> set = new HashSet<>();
set.addAll(Arrays.asList("one", "two")); //[one, two]
Set<String> copy = new HashSet<>(set);
<copy>.addAll(Arrays.asList("two", "three")); //[one, two, three]
<copy>.retainAll(Arrays.asList("two", "three")); //[two]
<copy>.removeAll(Arrays.asList("two", "three")); //[one]
```

Collections (maps)

```kotlin
val map = mapOf("one" to 1, "two" to 2) //{one=1, two=2}
map + mapOf("two" to 22, "three" to 3) //{one=1, two=22, three=3}
map - "one" //{two=2}
```

- copying and working with maps was cumbersome prior Java 9

```java
Map<String, Integer> map = new HashMap<>();
map.put("one", 1);
map.put("two", 2); //{one=1, two=2}
Map<String, Integer> otherMap = new HashMap<>();
otherMap.put("two", 22);
otherMap.put("three", 3);
Map<String, Integer> copy = new HashMap<>(map);
<copy>.putAll(otherMap); //{one=1, two=22, three=3}
<copy>.remove("one"); //{two=2}
```

Loops

```
val values = arrayOf(1, 2, 3, 4, 5)
for (i in values) {
   println(i)
}
```

- **for** can iterate through collections and iterables
- iterations will use values, not indexes (no need to reference items by indices)
- in case of maps iterations will use key-value pairs

```
val oneTwo = mapOf("one" to 1, "two" to 2)
for((k,v) in oneTwo) { println("$k -> $v") }
```

- **while** and **do-while** also exists and works the same way

```
int[] values = new int[]{1, 2, 3, 4, 5};
for (int i = 0; i < values.length; i++) {
   System.out.println(values[i]);
}
```

Ranges

```kotlin
for (i in 1..5) println(i)

if (3 in 1..5) println("3 is between 1 and 5")
```

- easy to create range of values
- ranges can also be used in for loops
- **`in`** within conditions will check if a range contains the given value
- ranges can be iterated in reverse order by using **`downTo`**

```kotlin
for (i in 5 downTo 1)
```

- **`step`** can be used to skip some values in iteration

```kotlin
for (i in 1..5 step 2)
```

```java
// same as before

if (1 <= 3 && 3 <= 5) System.out.println("3 is between 1 and 5");
```

Collection transformations

```
val result = (1..10) // 1 2 3 4 5 6 7 8 9 10
    .filter { it % 2 == 1 } // 1 3 5 7 9
    .map { it + 1 } // 2 4 6 8 10
    .takeLast(3) // 6 8 10
    .sum() //24
```

- a **lot** of transformation functions instantly available on collections (no **`stream()`** needed)
- every transformation will create a new collection
- no terminal operation required (i.e. **`collect(...)`**)
- most of the transformations are lambda functions **`{ }`**
- single input can be referred as **`it`** within lambda functions (**`n -> n`** becomes **`it`**)
- many transformations are much easier with Kotlin (i.e. groupings and working with maps)

```
int result = IntStream.range(1, 10)
        .filter(n -> n % 2 == 1)
        .map(n -> n + 1)
        .skip(2)
        .sum();
```

Classes

```
final public class Person {
   final private String name;

   public Person() {
       this.name = "John Doe";
   }

   public Person(String name) {
       if (name == null) {
           this.name = "John Doe";
       } else {
           this.name = name;
       }
   }

   public String getName() {
       return this.name;
   }
}

System.out.println("Hi, " + new Person().getName() +"!"); //Hi, John Doe!
```

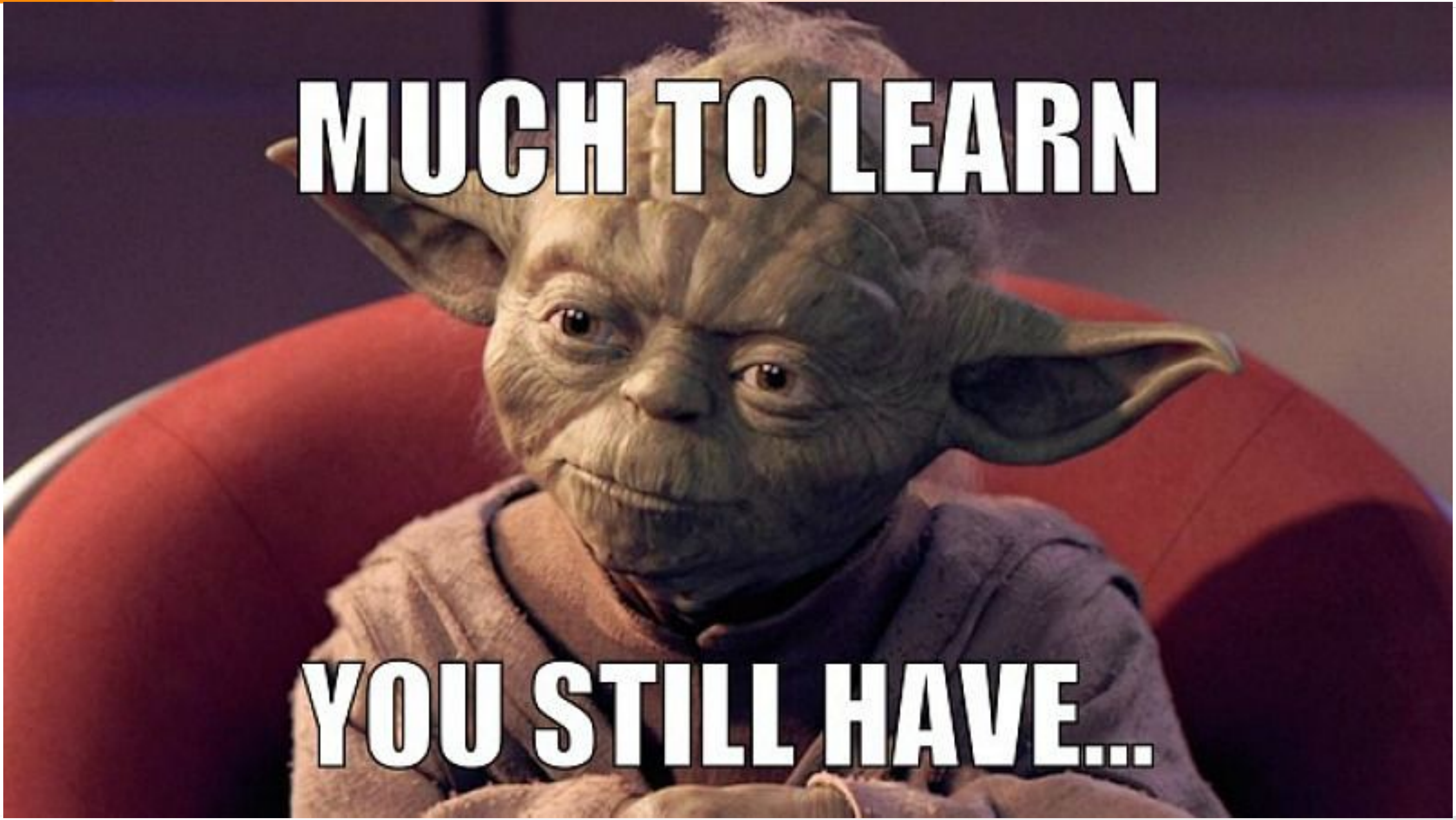
- final, public class with single property (immutable)
- constructor with default property value
- constructor to pass property value
- property getter
- print by creating new object and getting property

Classes

```kotlin
class Person(val name: String = "John Doe")

println("Hi, ${Person().name}!") //Hi, John Doe!
```

- this is the same class in Kotlin
- **`new`** keyword not needed when creating object
- access property directly
- constructors and getters are generated at bytecode (setters when class is mutable; no more Lombok)
- it is also possible to have custom constructor or initializer
- in Java usually we do not handle null or default values (in Kotlin it is easy)
- named arguments could be used (no builders needed)


MUCH TO LEARN
YOU STILL HAVE...

Demo

Resources

Demo: https://github.com/domahidizoltan/presentation-hello-kotlin

Kotlin language guide: https://kotlinlang.org/docs/reference/

Kotlin Playground: https://play.kotlinlang.org/

Dmitry Jemerov and Svetlana Isakova: Kotlin in Action