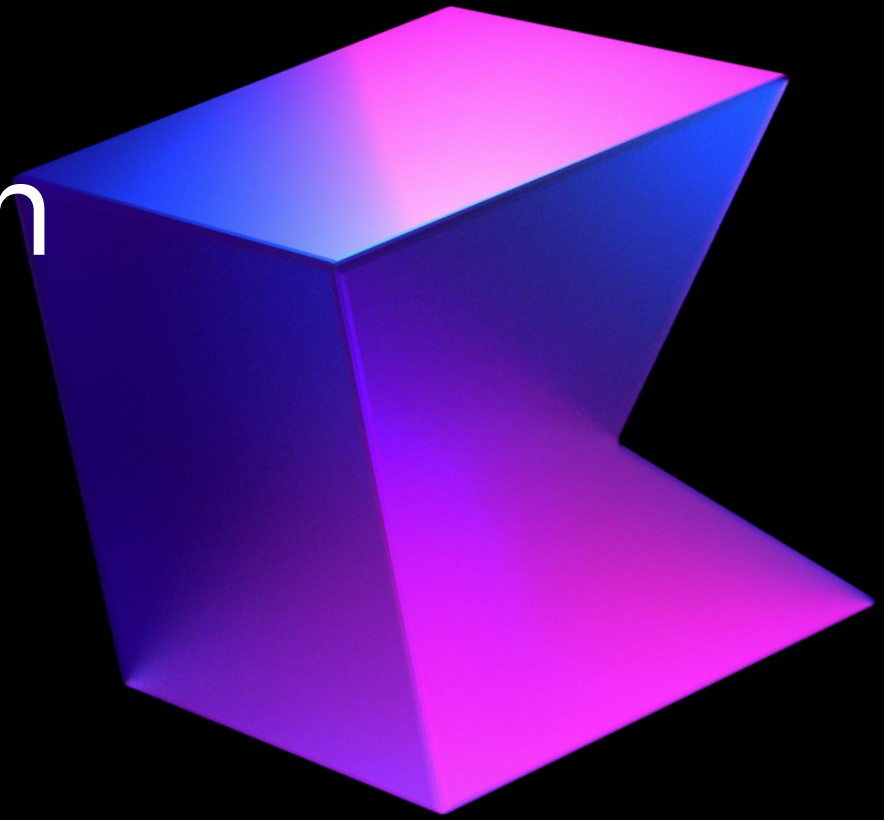
JET BRAINS Kotlin

Introduction to Kotlin

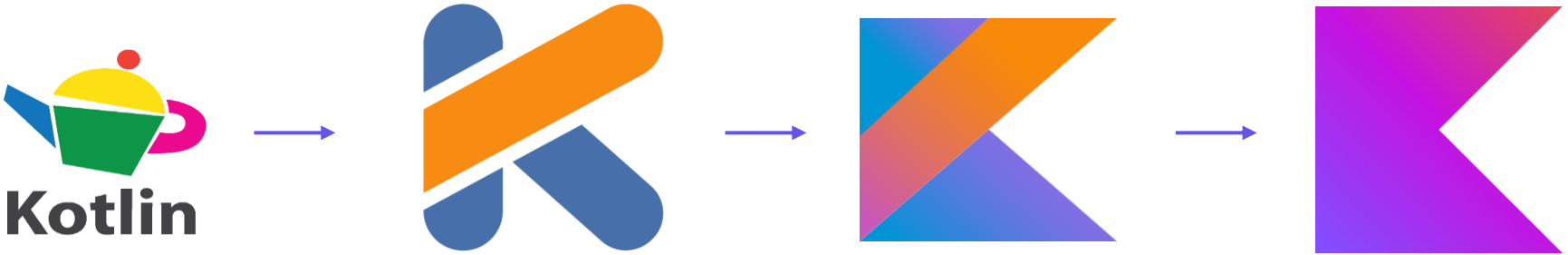


@kotlin | Developed by JetBrains

Why Kotlin?

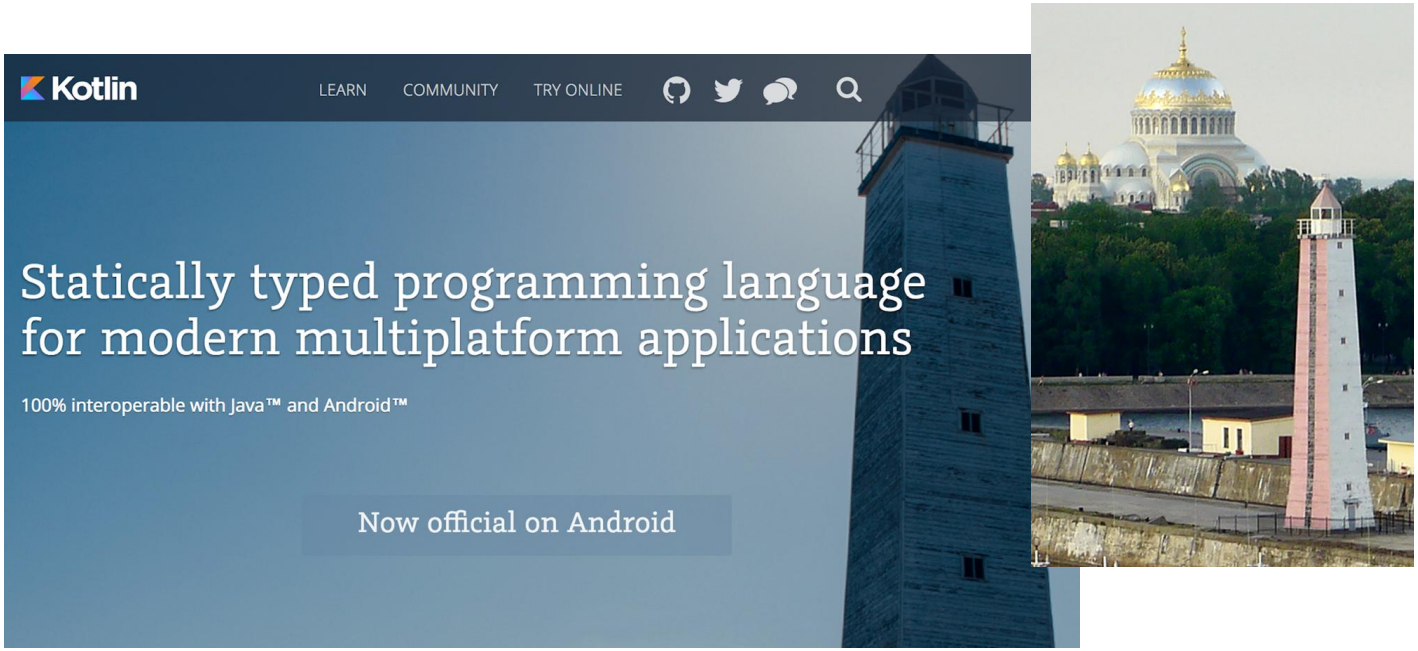
- Expressive and Concise
- Safer code (Null Safety)
- Portability / Compatibility
- Convenience
- High Quality IDE Support
- Community
- Android 👀
- More than a gazillion devices run ~~Java~~ Kotlin
- Lactose free
- ~~Sugar free~~
- Gluten free

Logo



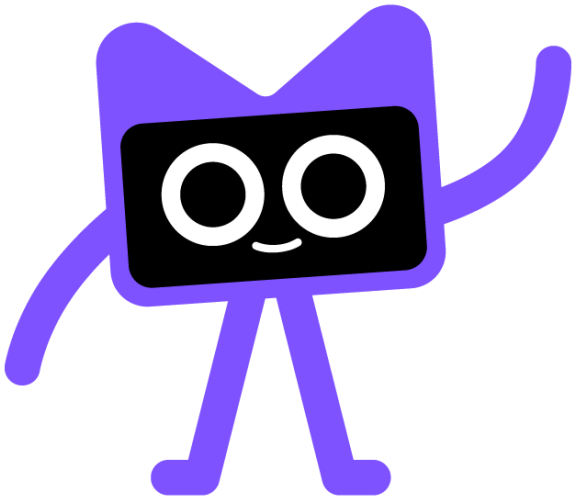
2011 2013 2016 2021

Name



Kotlin is named after an island in the Gulf of Finland.

Hello World



Hello, world!

```
fun main(args: Array<String>) {
    println("Hello, world!")
}

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

Where is “;”???

The basics

```
fun main(args: Array<String>) {
    print("Hello")
    println(", world!")
}
```

- An entry point of a Kotlin application is the `main` **top-level** function.
- It accepts a variable number of `String` arguments that can be omitted.
- `print` prints its argument to the standard output.
- `println` prints its arguments and adds a line break.

Kotlin Data Types

- Numbers and their unsigned counterparts
  - Signed Numbers: Byte, Int, Short, Long, Float, and Double
  - Unsigned Numbers: UByte, UInt, UShort, ULong
- Boolean
- Character
- String
- Arrays

Variables

`val/var myValue: Type = someValue`

- `var` - mutable
- `val` - immutable
- Type can be inferred in most cases
- Assignment can be deferred

```
val a: Int = 1              // immediate assignment

var b = 2                   // 'Int' type is inferred
b = a                       // Reassigning to 'var' is okay

val c: Int                  // Type required when no initializer is provided
c = 3                       // Deferred assignment
a = 4                       // Error: Val cannot be reassigned
```

Variables

```
const val/val myValue: Type = someValue
```

- `const val` - compile-time const value
- `val` - immutable value
- for `const val` use uppercase for naming

```
const val NAME = "Kotlin" // can be calculated at compile-time
val nameLowered = NAME.lowercase()  // cannot be calculated at compile-time
```

Identifiers

An identifier is the name given to a feature (variable, method, or class).

Names can contain letters, digits, underscores, and dollar signs

Names are case sensitive

Names can’t contain white spaces.

An identifier can begin with either:

1. Letter
2. $
3. Underscore.

Subsequent characters may be:

1. Letter
2. $
3. Underscore
4. Digits.

Array

Arrays in Kotlin has a **fixed** size, but its elements are **mutable**.

Array instances can be created using the **arrayOf( )**, **arrayOfNulls( )**, and **emptyArray( )** standard library functions.

```
val arrayOfNull = arrayOfNulls<Int>(20) //Both type and size must be known
val array = arrayOf(1,2,3,4,5,6)        //Must only define its contents during the compile-time
val emptyIntArray = emptyArray<Int>()   //Must only specify the type of data held by the array
```

We can combine two arrays and get the result array

```
val oddArray = arrayOf(1,3,5,7,9)
val evenArray = arrayOf(0,2,4,6,8)
val resultArray = oddArray + evenArray
```

Array

Kotlin also has classes that represent arrays of primitive **`ByteArray`**, **`ShortArray`**, **`IntArray`**, and so on.

```
val intArray : IntArray = intArrayOf(1,2,3,4,5,6)

val doubleArray : DoubleArray = doubleArrayOf(1.5, 2.7, 3.14)
```

Controlling Program Flow

If expression

```
val a = 20
val b = 10
val maximum = if (a > b) {
    println(“A is greater”)
    a
} else {
    println(“B is greater”)
    b
}
```

`if` can be an expression (it can return).
If you use `if` to get a result, `else` must be used; otherwise, it’s optional and could be omitted.

When expression

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

`when` returns, the same way that `if` does.

```
when {
    x < 0 -> print("x < 0")
    x > 0 -> print("x > 0")
    else -> {
        print("x == 0")
    }
}
```

The condition can be inside of the branches.

Loops (while & do while)

```
val items = arrayOf("apple", "banana", "kiwifruit")

var index = 0
while (index < items.size) {
    println("item at " + index + " is " items[index])
    index++
}

var flag : Boolean = true
do {
    ...
    flag= ...
} while(flag)
```

Loops (repeat)

Kotlin also has `repeat` loops that let you repeat a block of code that follows inside its curly braces. The number in parentheses is the number of times it should be repeated.

```
repeat(2){
    print("Hello!")
}

⇒ Hello!Hello!
```

Loops (for Loop)

```
val items = arrayOf("apple", "banana", "kiwifruit")

for (item in items) {
    println(item)
}

for (index in items.indices) {
    println("item at: " + index + " is: " + items[index])
}

for (index in 0 .. items.size) {
    println("item at: "+  index + " is: " + items[index])
}
```

Ranges

```
val x = 10
if (x in 1..10) {
    println("fits in range")
}

for (x in 1..5) {
    print(x)
}

for (x in 9 downTo 0 step 3) {
    print(x)
}
```

Labels

There are `break` and `continue` labels for loops:

```
myLabel@ for (item in items) {
    for (anotherItem in otherItems) {
        if (...) break@myLabel
        else continue@myLabel
    }
}
```

Strings

Strings

Generally, a string value is a sequence of characters in double quotes (" ")

```
val str = "abcd 123"
```

Elements of a string are characters that you can access via the indexing operation: s[i].
You can iterate over these characters with a for loop:

```
for (c in str) {
    println(c)
}
```

Strings are **immutable**.
All operations that transform strings return their results in a new String object,
leaving the original string unchanged

String templates

A template expression starts with a dollar sign ($) and consists of either **a variable name**:

```
val i = 10
val s = "Kotlin"

println("i = $i")
```

or an expression in curly braces:

```
println("Length of $s is ${s.length}")
```

String builder

String builder can be used to efficiently perform multiple string manipulation operations.

```
val sb = StringBuilder()
sb.append("Hello")
sb.append(", world!")
println(sb.toString())
```

Null Safety

Null safety

```
val notNullText: String = "Definitely not null"
val nullableText1: String? = "Might be null"
val nullableText2: String? = null
var text: String? = null
        if (text != null)
                println(text)
        else
                println("Nothing to print :(")
```

Safe Calls

`someThing?.otherThing` does **not** throw a NullPointerException if `someThing` is `null`.

Safe calls are useful in chains.
For example, an employee may be assigned to a department (or not).
That department may, in turn, have another employee as a department head,
who may or may not have a name, which we want to print:

```
println(employee.department?.head?.name)
```

To print only for non-null values, you can use the safe call operator together with `let`:

```
employee.department?.head?.name?.let { println(it) }
```

Unsafe Calls

The not-null assertion operator (`!!`) converts any value to a non-null type and throws a NPE exception if the value is null.

```
fun printDepartmentHead(employee: Employee) {
        println(employee.department!!.head!!.name!!)
}
```

Please, avoid using unsafe calls!

Elvis operator ?:

If the expression to the left of `?:` is not `null`, the Elvis operator returns it;
otherwise, it returns the expression to the right.

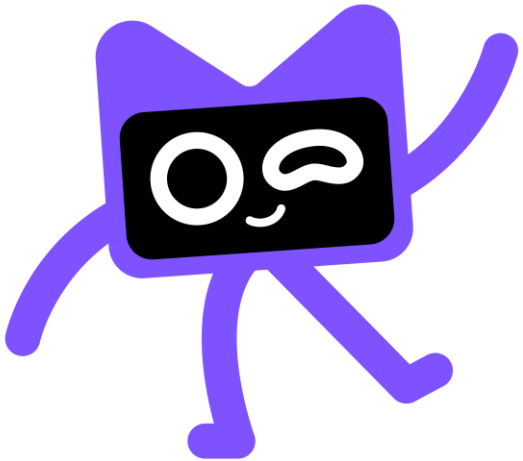
Note that the expression on the right-hand side is evaluated only if the left-hand side is `null`.

```
val id: String = “1234ABC”
val item = findItem(id) ?: return null
```

When in doubt

Go to:

- kotlinlang.org
- kotlinlang.org/docs
- play.kotlinlang.org/byExample



Lab

1) Use the readline() to read inputs from the user.
User should input their name, if user enters empty string, store a default value.
Use an if expression to define greeting message based on whether the user entered a name

2) create an Array of integer.
Use Random to fill the Array with 100 random number between 1 and 100.
Go through collection and print values less than or equal to 10

3) Calculator: ADD, SUB, MUL, DIV

Today’s Challenge