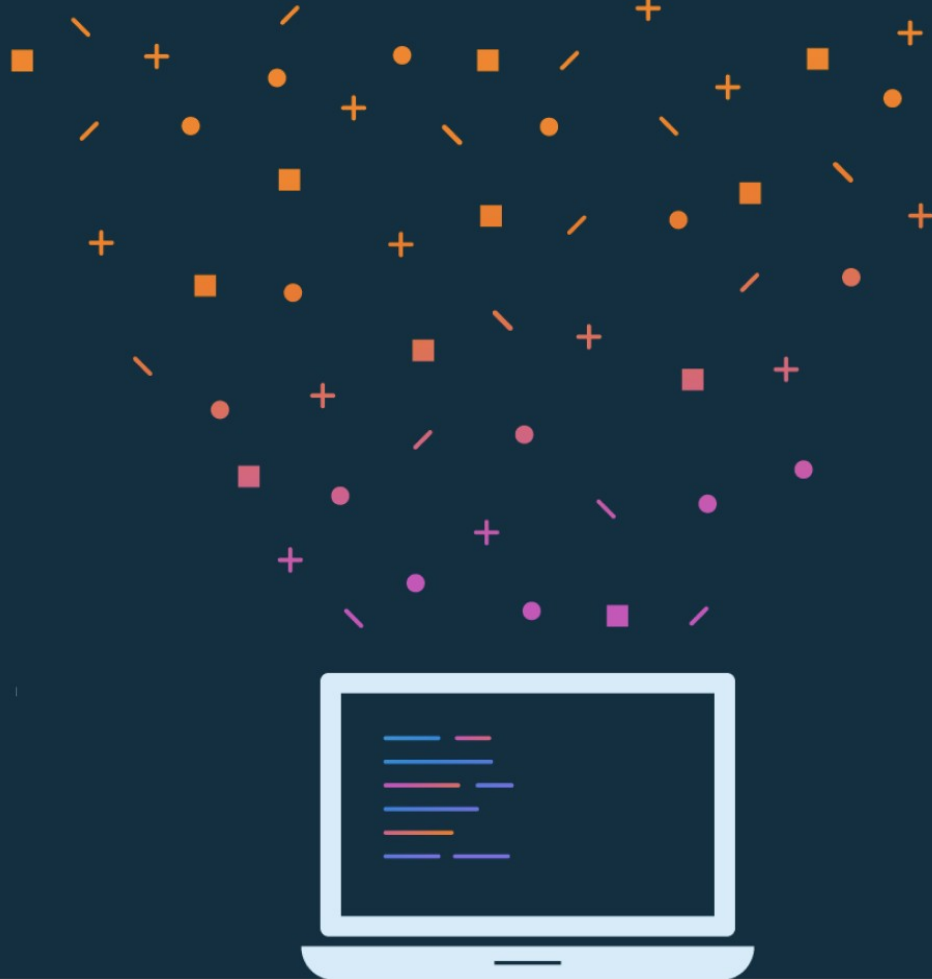


# Lesson 1: Kotlin basics



# About this lesson

## Lesson 1: Kotlin basics

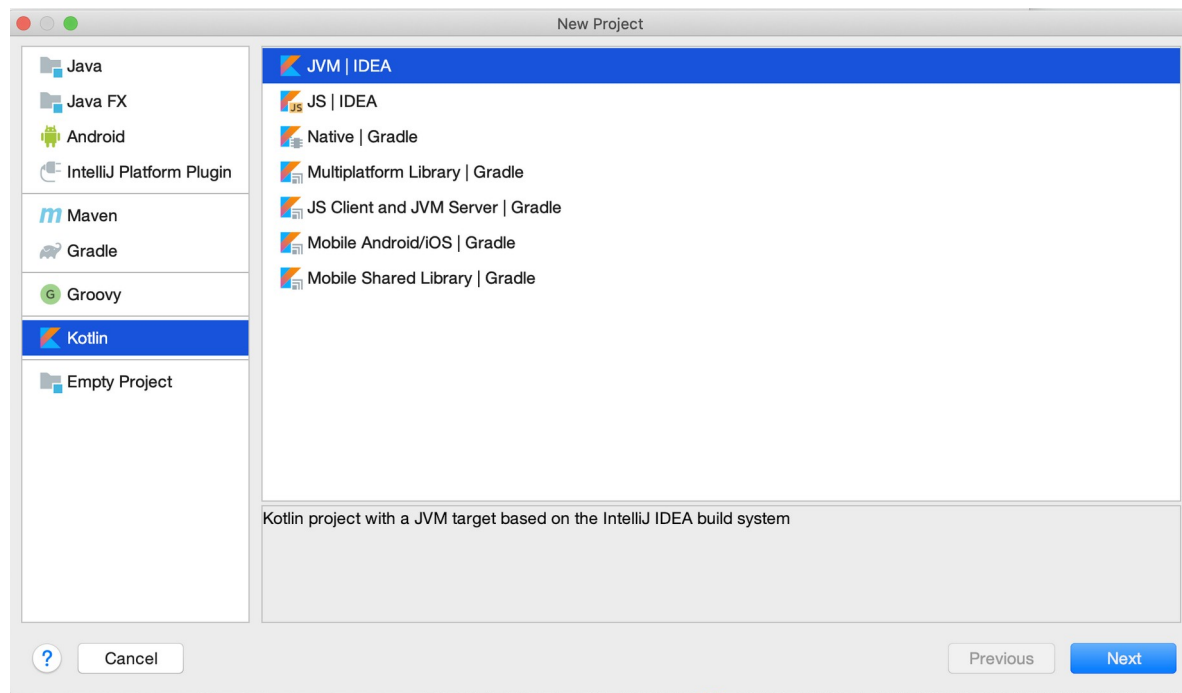
- [Get started](#)
- [Operators](#)
- [Data types](#)
- [Variables](#)
- [Conditionals](#)
- [Lists and arrays](#)
- [Null safety](#)
- [Summary](#)

# Get started

# Open IntelliJ IDEA



# Create a new project



# Name the project

New Project

Project name:

Project location:  ...

Project SDK:  New...

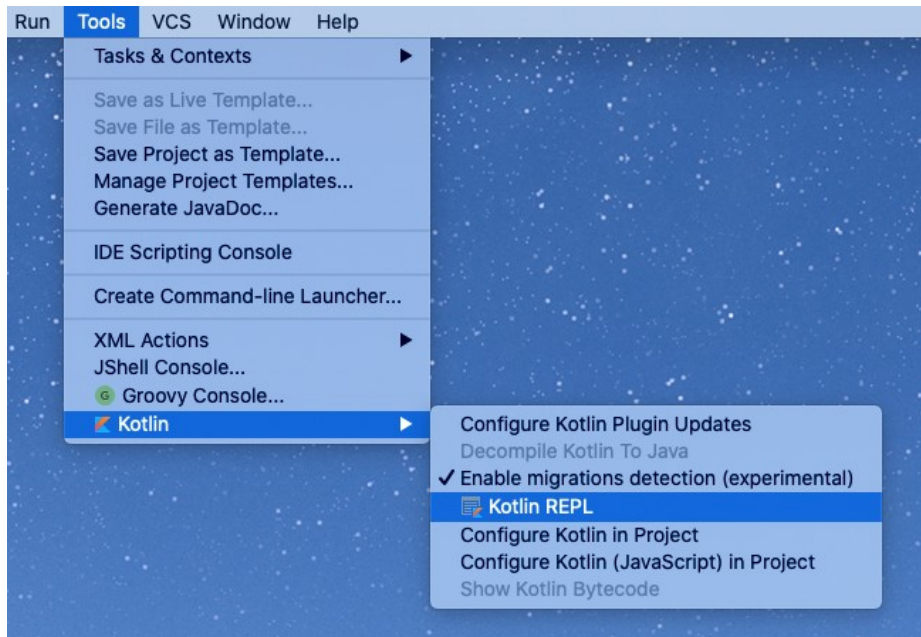
Kotlin runtime

Use library:  Create...

Project level library **KotlinJavaRuntime** with 3 files will be created

► More Settings

# Open REPL (Read-Eval-Print-Loop)



It may take a few moments before the Kotlin menu appears under

**Tools.**

# Create a printHello() function

```
Run: Kotlin REPL (in module HelloKotlin) x
Welcome to Kotlin version 1.3.41 (JRE 11.0.2+9-LTS)
Type :help for help, :quit for quit

fun printHello() {
    println("Hello World")
}

printHello()
Hello World

⌘↵ to execute
```

Press  
**Control+Enter**  
(**Command+Enter**  
on a Mac) to  
execute.



# Operators

# Operators

- Mathematical operators

+ - \* / %

- Increment and decrement operators

++ --

- Comparison operators

< <= > >=

- Assignment operator

=

- Equality operators

== !=

# Numeric operator methods

Kotlin keeps numbers as primitives, but lets you call methods on numbers as if they were objects.

```
2.times(3)
```

```
⇒ kotlin.Int = 6
```

```
3.5.plus(4)
```

```
⇒ kotlin.Double = 7.5
```

```
2.4.div(2)
```

```
⇒ kotlin.Double = 1.2
```

# Data types

# Integer types

Type	Bits	Notes
Long	64	From $-2^{63}$ to $2^{63}-1$
Int	32	From $-2^{31}$ to $2^{31}-1$
Short	16	From -32768 to 32767
Byte	8	From -128 to 127

# Floating-point and other

Type	Bits	Notes
Double	64	16 - 17 significant digits
Float	32	6 - 7 significant digits
Char	16	16-bit Unicode character
Boolean	8	True or false. Operations include:    - lazy disjunction, && - lazy conjunction, ! - negation



# Operand types

Results of operations keep the types of the operands

$6 * 50$

⇒ `kotlin.Int = 300`

$1/2$

⇒ `kotlin.Int = 0`

$6.0 * 50.0$

⇒ `kotlin.Double = 300.0`

$1.0 * 2.0$

⇒ `kotlin.Double = 0.5`

$6.0 * 50$

⇒ `kotlin.Double = 300.0`

# Type casting

Assign an `Int` to a `Byte`

```
val i: Int = 6  
val b: Byte = i  
println(b)
```

⇒ error: type mismatch: inferred type is Int but Byte was expected

Convert `Int` to `Byte` with casting

```
val i: Int = 6  
println(i.toByte())
```

⇒ 6



# Underscores for long numbers

Use underscores to make long numeric constants more readable.

```
val oneMillion = 1_000_000
```

```
val idNumber = 999_99_9999L
```

```
val hexBytes = 0xFF_EC_DE_5E
```

```
val bytes = 0b11010010_01101001_10010100_10010010
```

# Strings

Strings are any sequence of characters enclosed by double quotes.

```
val s1 = "Hello world!"
```

String literals can contain escape characters

```
val s2 = "Hello world!\n"
```

Or any arbitrary text delimited by a triple quote ("""")

```
val text = """  
var bikes = 50  
"""
```

# String concatenation

```
val numberOfDogs = 3
```

```
val numberOfCats = 2
```

```
"I have $numberOfDogs dogs" + " and $numberOfCats cats"
```

```
=> I have 3 dogs and 2 cats
```

# String templates

A template expression starts with a dollar sign (\$) and can be a simple value:

```
val i = 10  
println("i = $i")
```

=> i = 10

Or an expression inside curly braces:

```
val s = "abc"  
println("$s.length is ${s.length}")  
=> abc.length is 3
```

# String template expressions

```
val numberOfShirts = 10
```

```
val numberOfPants = 5
```

```
"I have ${numberOfShirts + numberOfPants} items of clothing"
```

```
=> I have 15 items of clothing
```

# Variables

# Variables

- Powerful type inference
  - Let the compiler infer the type
  - You can explicitly declare the type if needed
- Mutable and immutable variables
  - Immutability not enforced, but recommended

Kotlin is a statically-typed language. The type is resolved at compile time and never changes.

# Specifying the variable type

## Colon Notation

```
var width: Int = 12  
var length: Double = 2.5
```

**Important:** Once a type has been assigned by you or the compiler, you can't change the type or you get an error.



# Mutable and immutable variables

- Mutable (Changeable)

```
var score = 10
```

- Immutable (Unchangeable)

```
val name = "Jennifer"
```

Although not strictly enforced, using immutable variables is recommended in most cases.

# var and val

```
var count = 1
```

```
count = 2
```

```
val size = 1
```

```
size = 2
```

```
=> Error: val cannot be reassigned
```

# Conditionals

# Control flow

Kotlin features several ways to implement conditional logic:

- If/Else statements
- When statements
- For loops
- While loops

# if/else statements

```
val numberOfCups = 30
val numberOfPlates = 50

if (numberOfCups > numberOfPlates) {
    println("Too many cups!")
} else {
    println("Not enough cups!")
}

=> Not enough cups!
```

# if statement with multiple cases

```
val guests = 30
if (guests == 0) {
    println("No guests")
} else if (guests < 20) {
    println("Small group of people")
} else {
    println("Large group of people!")
}
```

⇒ Large group of people!

# Ranges

- Data type containing a span of comparable values (e.g., integers from 1 to 100 inclusive)
- Ranges are bounded
- Objects within a range can be mutable or immutable

# Ranges in if/else statements

```
val numberOfStudents = 50
if (numberOfStudents in 1..100) {
    println(numberOfStudents)
}
=> 50
```

**Note:** There are no spaces around the "range to" operator (1..100)



# when statement

```
when (results) {  
    0 -> println("No results")  
    in 1..39 -> println("Got results!")  
    else -> println("That's a lot of results!")  
}  
⇒ That's a lot of results!
```

As well as a `when` statement, you can also define a `when` expression that provides a return value.

# for loops

```
val pets = arrayOf("dog", "cat", "canary")  
for (element in pets) {  
    print(element + " ")  
}  
⇒ dog cat canary
```

You don't need to define an iterator variable and increment it for each pass.

# for loops: elements and indexes

```
for ((index, element) in pets.withIndex()) {  
    println("Item at $index is $element\n")  
}
```

⇒ Item at 0 is dog

Item at 1 is cat

Item at 2 is canary

# for loops: step sizes and ranges

```
for (i in 1..5) print(i)
```

⇒ 12345

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

⇒ 54321

```
for (i in 3..6 step 2) print(i)
```

⇒ 35

```
for (i in 'd'..'g') print (i)
```

⇒ defg

# while loops

```
var bicycles = 0
while (bicycles < 50) {
    bicycles++
}
```

```
println("$bicycles bicycles in the bicycle rack\n")
⇒ 50 bicycles in the bicycle rack
```

```
do {
    bicycles--
} while (bicycles > 50)
```

```
println("$bicycles bicycles in the bicycle rack\n")
⇒ 49 bicycles in the bicycle rack
```

# repeat loops

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

⇒ Hello!Hello!

# Lists and arrays

# Lists

- Lists are ordered collections of elements
- List elements can be accessed programmatically through their indices
- Elements can occur more than once in a list

An example of a list is a sentence: it's a group of words, their order is important, and they can repeat.



# Immutable list using listOf()

Declare a list using `listOf()` and print it out.

```
val instruments = listOf("trumpet", "piano", "violin")  
println(instruments)
```

```
⇒ [trumpet, piano, violin]
```

# Mutable list using mutableListOf()

Lists can be changed using `mutableListOf()`

```
val myList = mutableListOf("trumpet", "piano", "violin")  
myList.remove("violin")  
  
⇒ kotlin.Boolean = true
```

With a list defined with `val`, you can't change which list the variable refers to, but you can still change the contents of the list.

# Arrays

- Arrays store multiple items
- Array elements can be accessed programmatically through their indices
- Array elements are mutable
- Array size is fixed

# Array using arrayOf()

An array of strings can be created using `arrayOf()`

```
val pets = arrayOf("dog", "cat", "canary")  
println(java.util.Arrays.toString(pets))  
⇒ [dog, cat, canary]
```

With an array defined with `val`, you can't change which array the variable refers to, but you can still change the contents of the array.

# Arrays with mixed or single types

An array can contain different types.

```
val mix = arrayOf("hats", 2)
```

An array can also contain just one type (integers in this case).

```
val numbers = intArrayOf(1, 2, 3)
```

# Combining arrays

Use the + operator.

```
val numbers = intArrayOf(1,2,3)
val numbers2 = intArrayOf(4,5,6)
val combined = numbers2 + numbers
println(Arrays.toString(combined))
```

```
=> [4, 5, 6, 1, 2, 3]
```

# Null safety

# Null safety

- In Kotlin, variables cannot be null by default
- You can explicitly assign a variable to null using the safe call operator
- Allow null-pointer exceptions using the `!!` operator
- You can test for null using the elvis (`? :`) operator



# Variables cannot be null

In Kotlin, `null` variables are not allowed by default.

Declare an `Int` and assign `null` to it.

```
var numberOfBooks: Int = null
```

⇒ error: null can not be a value of a non-null type  
`Int`

# Safe call operator

The safe call operator (?), after the type indicates that a variable can be `null`.

Declare an `Int?` as nullable

```
var numberOfBooks: Int? = null
```

In general, do not set a variable to null as it may have unwanted consequences.

# Testing for null

Check whether the `numberOfBooks` variable is not `null`. Then decrement that variable.

```
var numberOfBooks = 6
if (numberOfBooks != null) {
    numberOfBooks = numberOfBooks.dec()
}
```

Now look at the Kotlin way of writing it, using the safe call operator.

```
var numberOfBooks = 6
numberOfBooks = numberOfBooks?.dec()
```

# The !! operator

If you're certain a variable won't be null, use !! to force the variable into a non-null type. Then you can call methods/properties on it.

```
val len = s!!.length
```



*throws NullPointerException if s is null*

**Warning:** Because !! will throw an exception, it should only be used when it would be exceptional to hold a null value.

# Elvis operator

Chain null tests with the `?:` operator.

```
numberOfBooks = numberOfBooks?.dec() ?: 0
```

The `?:` operator is sometimes called the "Elvis operator," because it's like a smiley on its side with a pompadour hairstyle, like Elvis Presley styled his hair.

# Summary

# Summary

In Lesson 1, you learned how to:

- Create an IntelliJ IDEA project, opening REPL, and execute a function
- Use operators and numeric operator methods
- Use data types, type casting, strings, and string templates
- Use variables and type inference, and mutable and immutable variables
- Use conditionals, control flow, and looping structures
- Use lists and arrays
- Use Kotlin's null safety features

# Pathway

Practice what you've learned by completing the pathway:

[Lesson 1: Kotlin basics](#)

