Developer Student Clubs

Kotlin Basics

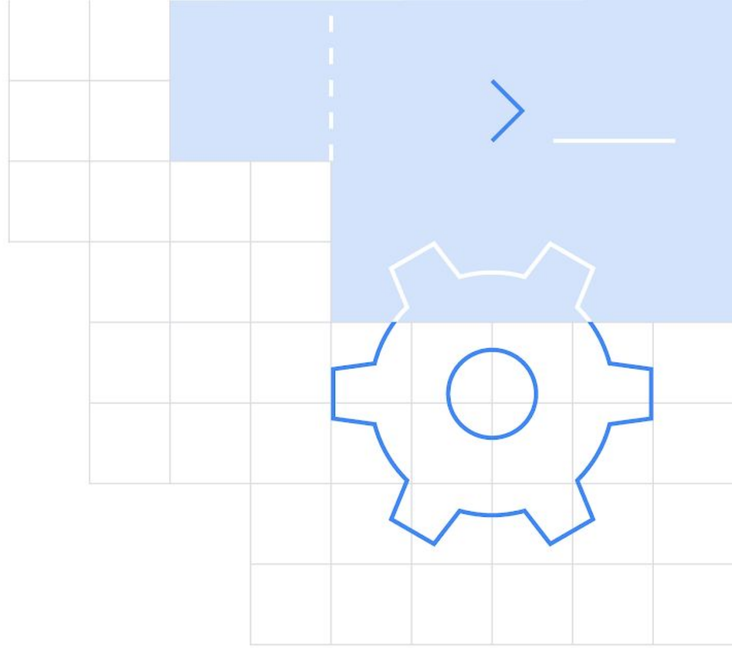
An introductory course to the Kotlin programming language

Stelios Papamichail
Android Engineer
@MikePapamichail

Google Developers

Agenda

- Basic Syntax
- Types
- Variables
- Functions
- Classes & Objects
- If-else expressions
- Loops
- The when() expression
- Ranges
- Collections
- and finally nullable values!



Developer Student Clubs

Getting Started

A modern & mature Java alternative

Kotlin is a modern but already mature programming language aimed to make developers happier. It's concise, safe, interoperable with Java and other languages, and provides many ways to reuse code between multiple platforms for productive programming.

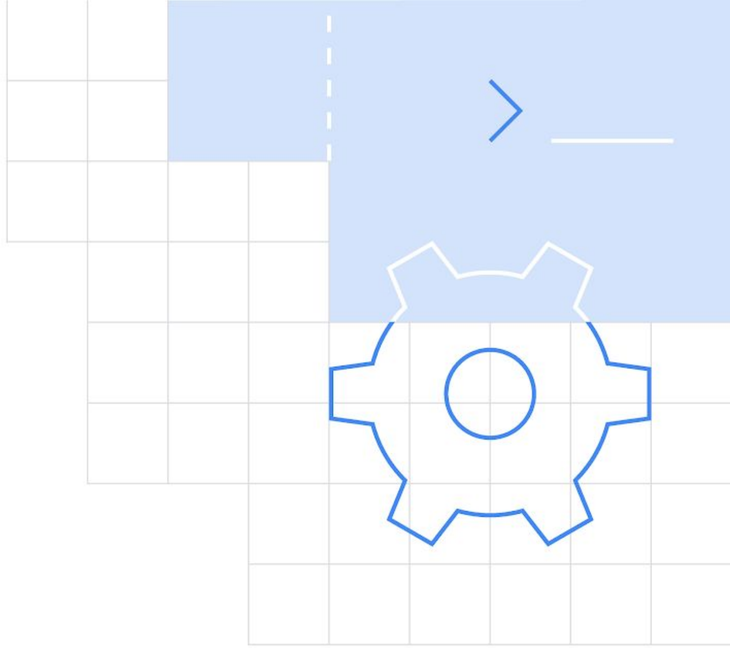
Kotlin is included in each IntelliJ IDEA and Android Studio release.
Download and install one of these IDEs to start using Kotlin or play around at the Kotlin Playground.

Building Powerful Apps

Creating software using Kotlin

Here are some examples of the different types of software that you can develop using Kotlin:

- Server-side apps using Kotlin for the backend
- Cross-platform mobile apps using KMM
- Web-app front-end thanks to Kotlin ↔ JS conversion
- Native Android apps (Kotlin is the recommended way)
- Multiplatform library development



Basic Syntax

Packages, Entry point, Types & Variables

Developer Student Clubs

Packages & Imports

Package specification should be at the top of the source file. It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

```
package my.demo

import kotlin.text.*

// ...
```

The main() function

A look at Kotlin's program entry point

An entry point to a Kotlin application is the main function (just like Java) and it usually accepts a variable number of String arguments.

Since Kotlin 1.3, you can declare main without any parameters. The return type is not specified, which means that the function returns nothing.

```kotlin
// no arguments
fun main() {
    println("Hello world!")
}

// variable string args
fun main(args: Array<String>) {
    println(args.contentToString())
}
```

Variables

Read-only, read/write & global variables

- **Read-only** local variables are defined using the keyword val. They can be assigned a value only once.
- Variables that can be **reassigned** use the var keyword.
- You can declare variables at the top level.

```
// read only variables
val a: Int = 1  // immediate assignment
val b = 2   // `Int` type is inferred
val c: Int  // Type required when no initializer is provided
c = 3       // deferred assignment
```

```
// normal variables
var x = 5 // `Int` type is inferred
x += 1
```

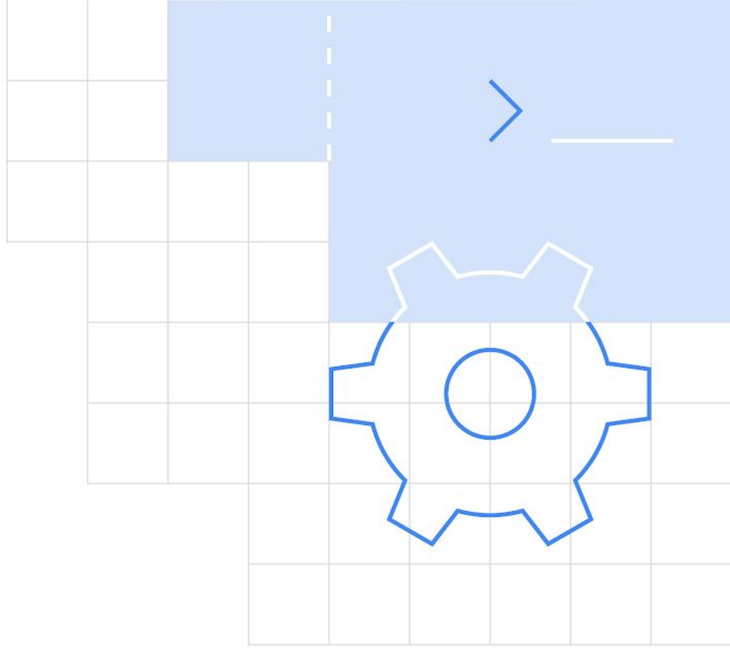
```
// top-level variables
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
```

Basic Data Types

Exploring the special nature of Kotlin's primitives

In Kotlin, **everything is an object** in the sense that you can call member functions and properties on any variable. Some types can have a special internal representation – for example, numbers, characters and booleans, can be represented as primitive values at runtime – but to the user they look like ordinary classes.

We will briefly cover the following basic types: **Numbers** & their unsigned counterparts, **Booleans**, **Characters**, **Strings** & **Arrays**!



Signed Integer Types

Developer Student Clubs

Integer Types

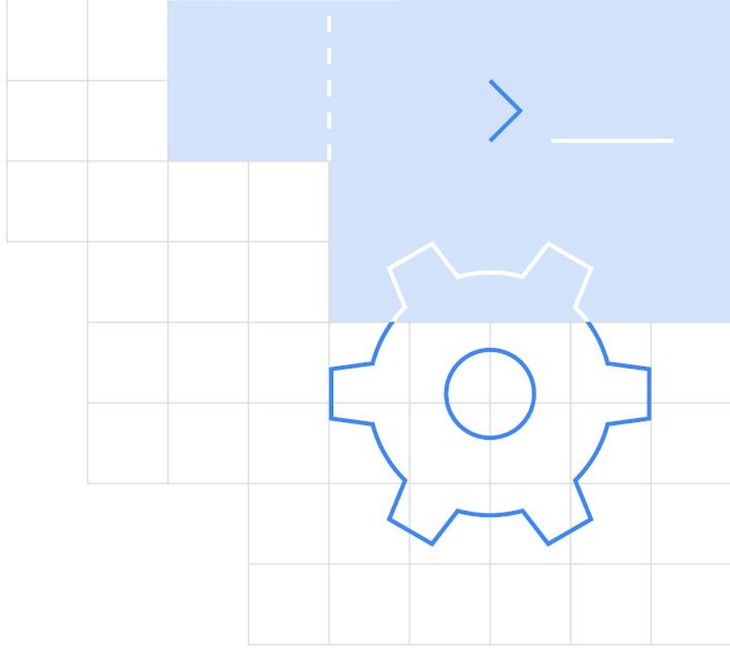
Kotlin provides a set of built-in types that represent numbers. For integer numbers, there are four types with different sizes and, hence, value ranges:

Type	Size (bits)	Min value	Max value
`Byte`	8	-128	127
`Short`	16	-32768	32767
`Int`	32	-2,147,483,648 (-2^{31})	2,147,483,647 ($2^{31} - 1$)
`Long`	64	-9,223,372,036,854,775,808 (-2^{63})	9,223,372,036,854,775,807 ($2^{63} - 1$)

Integer Types

When you initialize a variable with no explicit type specification, the compiler automatically infers the type with the smallest range enough to represent the value. If it is not exceeding the range of Int, the type is Int. If it exceeds, the type is Long. To specify the Long value explicitly, append the suffix L to the value. Explicit type specification triggers the compiler to check the value not to exceed the range of the specified type.

```
val one = 1 // Int
val threeBillion = 3000000000 // Long
val oneLong = 1L // Long
val oneByte: Byte = 1
```



Unsigned Integers

Developer Student Clubs

Unsigned Integer Types

In addition to integer types, Kotlin provides the following types for unsigned integer numbers:

UByte: an unsigned 8-bit integer, ranges from 0 to 255
UShort: an unsigned 16-bit integer, ranges from 0 to 65535
UInt: an unsigned 32-bit integer, ranges from 0 to 2^32 - 1
ULong: an unsigned 64-bit integer, ranges from 0 to 2^64 - 1

Unsigned types support most of the operations of their signed counterparts.

Use cases & non-goals

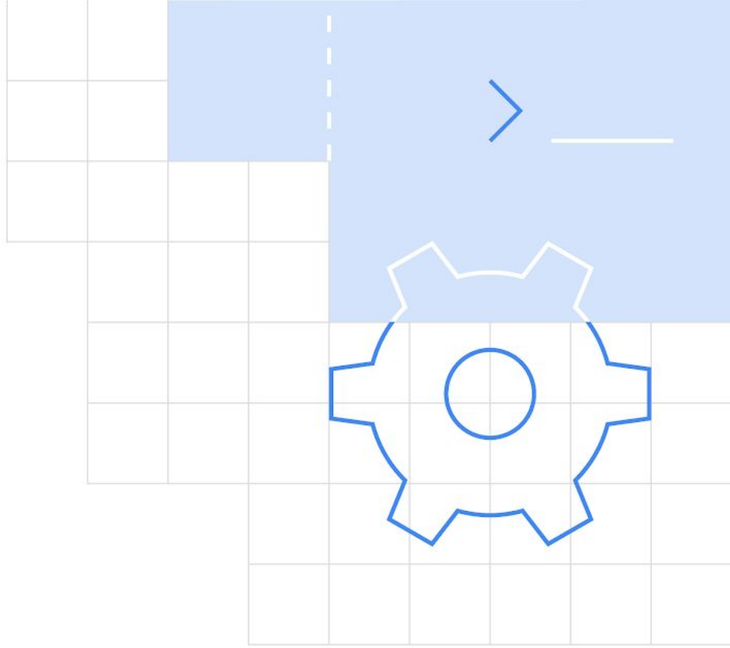
Use Case(s)

The main use case of unsigned numbers is utilizing the full bit range of an integer to represent positive values.
For example, to represent hexadecimal constants that do not fit in signed types such as color in 32-bit AARRGGBB format.

Non Goals

While unsigned integers can only represent positive numbers and zero, *it's not a goal to use them where the application domain requires non-negative integers*. For example, as a type of collection size or collection index value.
There are a couple of reasons:

- Using signed integers can help to detect accidental overflows and signal error conditions, such as List.lastIndex being -1 for an empty list.
- Unsigned integers cannot be treated as a range-limited version of signed ones because their range of values is not a subset of the signed integers range. Neither signed, nor unsigned integers are subtypes of each other.



Floating Point Types

Developer Student Clubs

Floating Point Types

For real numbers, Kotlin provides floating-point types Float and Double that adhere to the IEEE 754 standard. Float reflects the IEEE 754 *single precision*, while Double reflects *double precision*. These types differ in their size and provide storage for floating-point numbers with different precision:

Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
`Float`	32	24	8	6-7
`Double`	64	53	11	15-16

- You can initialize Double and Float variables with numbers having a fractional part. It's separated from the integer part by a period (.) For variables initialized with fractional numbers, the compiler infers the Double type:

```
val pi = 3.14 // Double
// val one: Double = 1 // Error: type mismatch
val oneDouble = 1.0 // Double
```

- To explicitly specify the Float type for a value, add the suffix f or F. If such a value contains more than 6-7 decimal digits, it will be rounded:

```
val e = 2.7182818284 // Double
val eFloat = 2.7182818284f // Float, actual value is 2.7182817
```

- Unlike some other languages, there are no implicit widening conversions for numbers in Kotlin. For example, a function with a Double parameter can be called only on Double values, but not Float, Int, or other numeric values:

```
fun main() {
    fun printDouble(d: Double) { print(d) }

    val i = 1
    val d = 1.0
    val f = 1.0f

    printDouble(d)
//    printDouble(i) // Error: Type mismatch
//    printDouble(f) // Error: Type mismatch
}
```

Literal Number Constants

There are the following kinds of literal constants for integral values:

- **Decimals**: 123
  - Longs are tagged by a capital L: 123L
- **Hexadecimals**: 0x0F
- **Binaries**: 0b00001011

⟶ Octal literals are not supported in Kotlin.

Kotlin also supports a conventional notation for floating-point numbers:

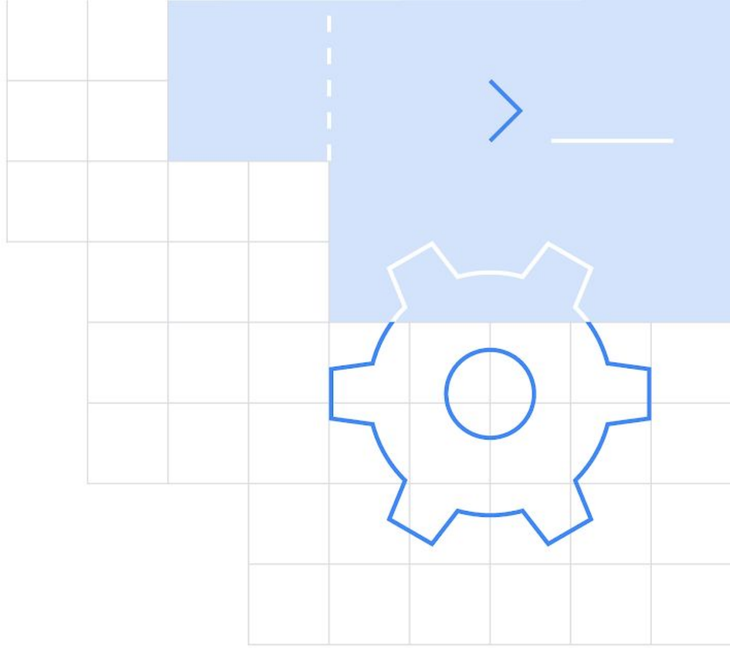
- **Doubles** by default: 123.5, 123.5e10
- **Floats** are tagged by f or F: 123.5f

- You can use underscores to make number constants more readable:

```
val oneMillion = 1_000_000
val creditCardNumber = 1234_5678_9012_3456L
val socialSecurityNumber = 999_99_9999L
val hexBytes = 0xFF_EC_DE_5E
val bytes = 0b11010010_01101001_10010100_10010010
```

- Kotlin supports the standard set of arithmetical operations over numbers: +, -, *, /, %. They are declared as members of appropriate classes:

```
println(1 + 2)
println(2_500_000_000L - 1L)
println(3.14 * 2.71)
println(10.0 / 3)
```



The Boolean type

Developer Student Clubs

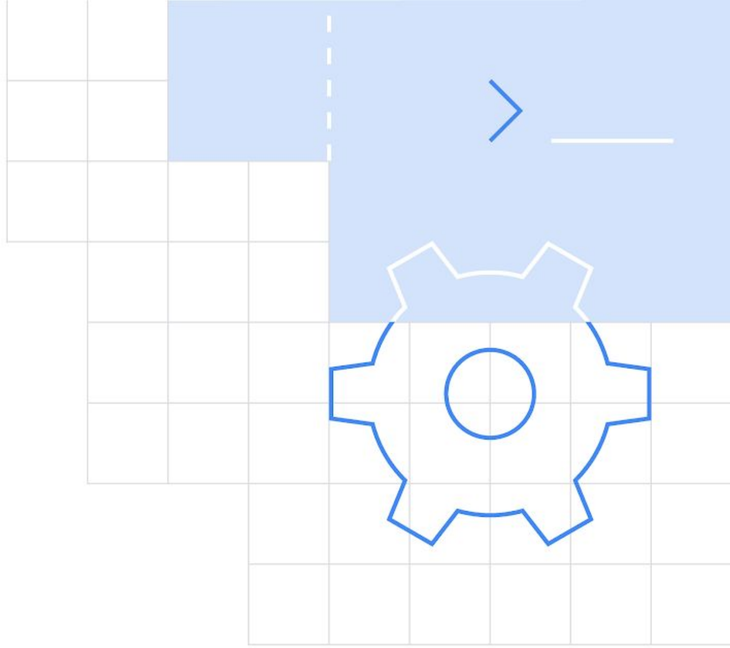
Boolean Types

The type Boolean represents boolean objects that can have two values: true and false (*Boolean has a nullable counterpart Boolean? that also has the null value*).
Built-in operations on booleans include:

- || – disjunction (logical *OR*)
- && – conjunction (logical *AND*)
- ! – negation (logical *NOT*)

|| and && work lazily.

```kotlin
val myTrue: Boolean = true
val myFalse: Boolean = false
val boolNull: Boolean? = null

println(myTrue || myFalse)
println(myTrue && myFalse)
println(!myTrue)
```



The Character type

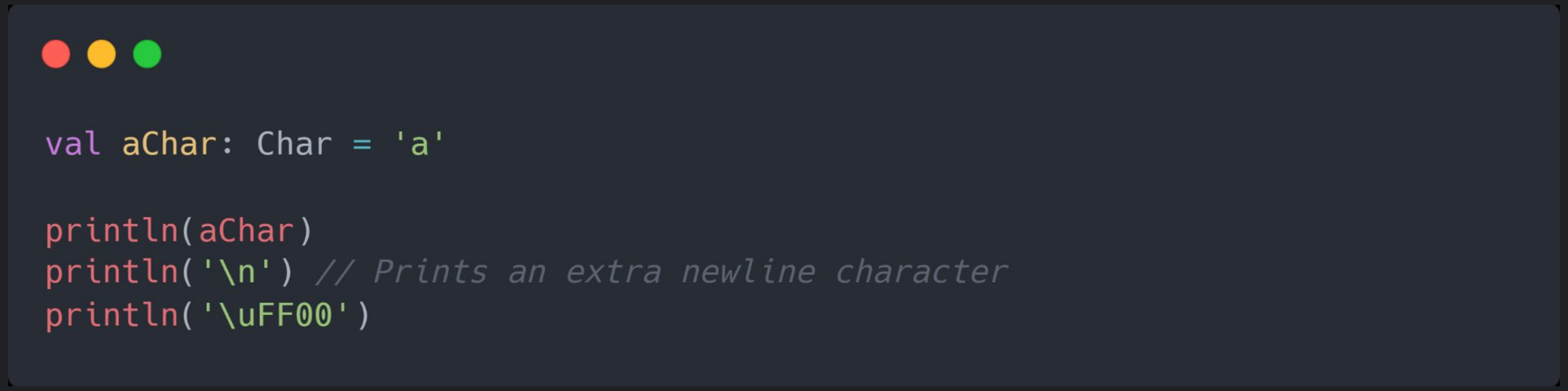
Developer Student Clubs

Character Type

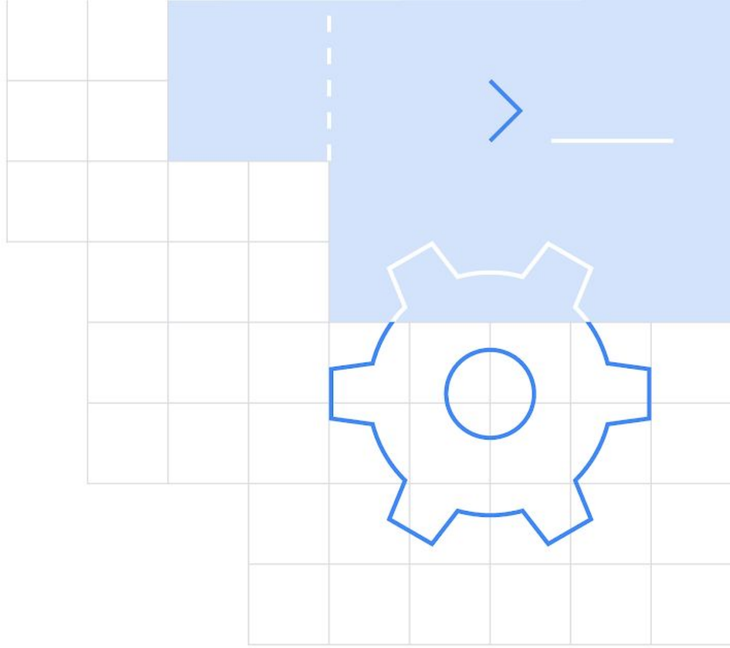
Characters are represented by the type Char & character literals go in single quotes: '1'.
Special characters start from an escaping backslash \. The following escape sequences are supported:

- \t – tab
- \b – backspace
- \n – new line (LF)
- \r – carriage return (CR)
- \' – single quotation mark
- \" – double quotation mark
- \\ – backslash
- \$ – dollar sign

To encode any other character, use the Unicode escape sequence syntax: '\uFF00'.



```
val aChar: Char = 'a'

println(aChar)
println('\n') // Prints an extra newline character
println('\uFF00')
```



Strings & String literals

Developer Student Clubs

String Type

Strings in Kotlin are represented by the type String.

Generally, a string value is a sequence of characters in double quotes (").
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. **Strings are immutable**.

Once you initialize a string, you can't change its value or assign a new value to it. All operations that transform strings return their results in a new String object, leaving the original string unchanged.

To concatenate strings, use the **+** operator. This also works for concatenating strings with values of other types, as long as the first element in the expression is a string.

```
val str = "abcd 123" // simple string

// iterate over str chars
for (c in str) {
    println(c)
}

// immutability example
val str = "abcd"
println(str.uppercase()) // Create and print a new String object
println(str) // The original string remains the same

// string concatenation
val s = "abc" + 1
println(s + "def")
```

String Literals

Kotlin has two types of string literals:

- Escaped strings
- Raw strings

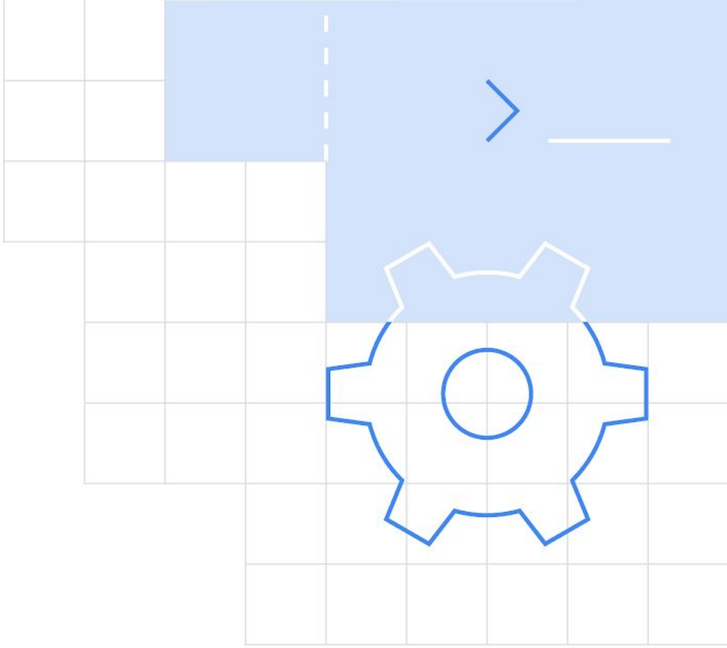
Escaped strings can contain escaped characters & escaping is done in the conventional way, with a backslash (\).

Raw strings can contain newlines and arbitrary text. It is delimited by a triple quote (""""), contains no escaping and can contain newlines and any other characters.

To remove leading whitespace from raw strings, use the trimMargin() function. By default, a pipe symbol | is used as margin prefix, but you can choose another character and pass it as a parameter, like trimMargin(">").

```
val s = "Hello, world!\n" // simple escaped string

// sample raw string
val text = """
    for (c in "foo")
        print(c)
"""

// trim margin example
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

String Templates



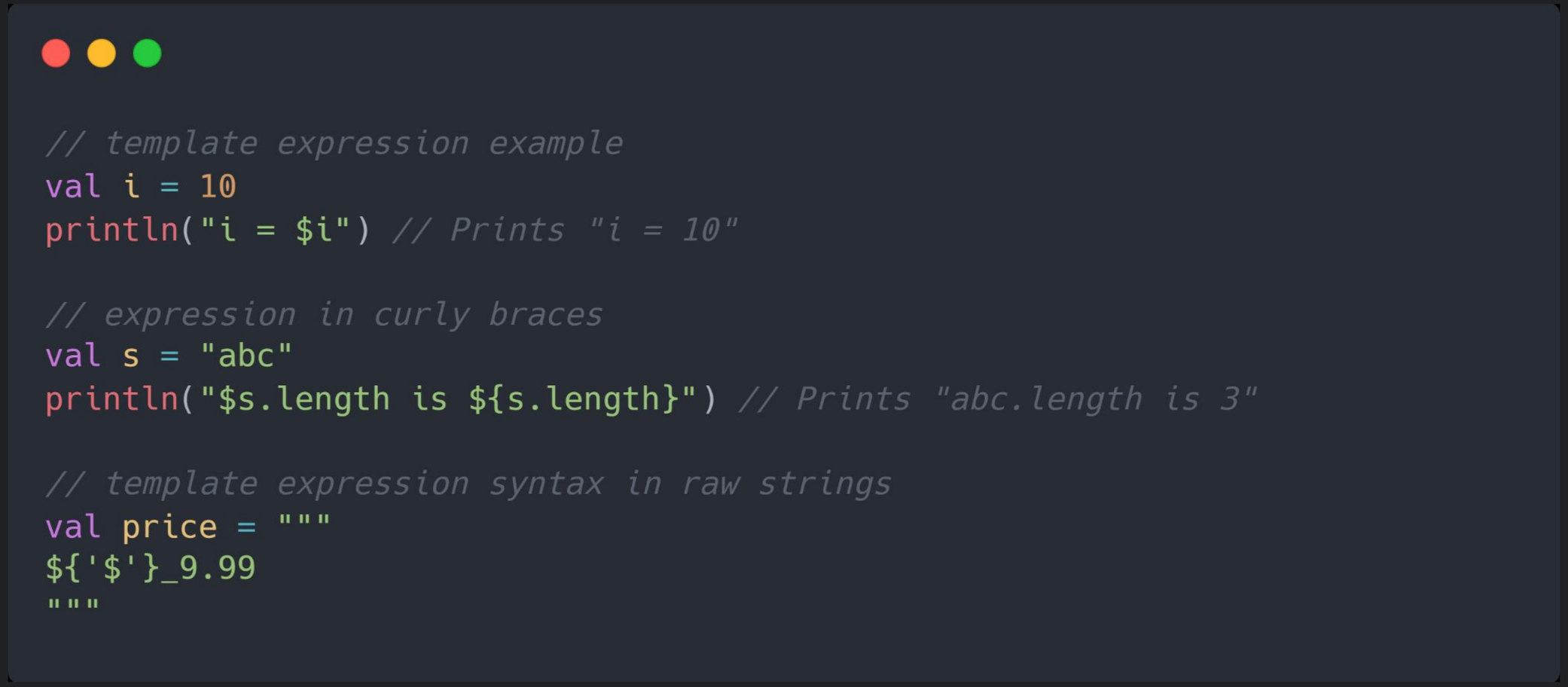
Developer Student Clubs

String Templates

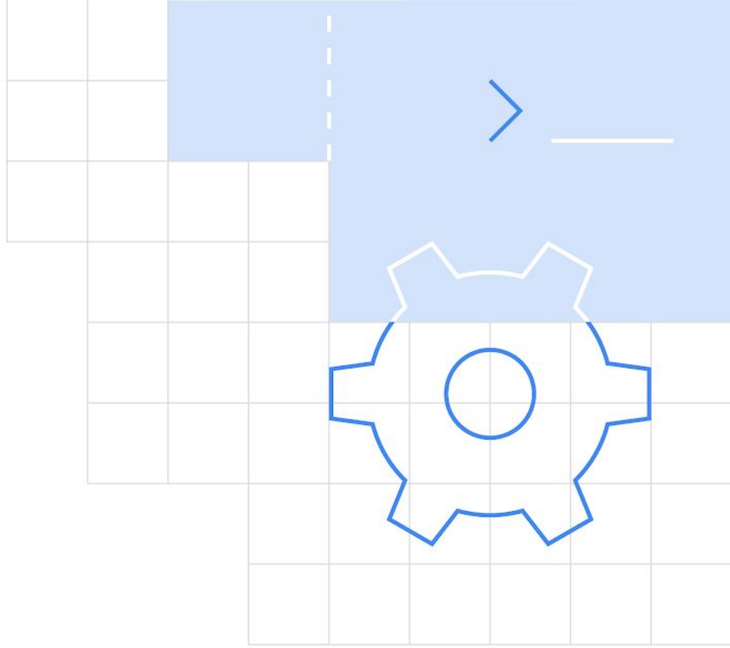
String literals may contain *template expressions* – pieces of code that are evaluated and whose results are concatenated into the string.

A template expression starts with a dollar sign ($) and consists of either a name or an expression in curly braces.

You can use templates both in raw and escaped strings. To insert the dollar sign $ in a raw string (which doesn't support backslash escaping) before any symbol, which is allowed as a beginning of an identifier, use the syntax shown in the following slide.



```
// template expression example
val i = 10
println("i = $i") // Prints "i = 10"

// expression in curly braces
val s = "abc"
println("$s.length is ${s.length}") // Prints "abc.length is 3"

// template expression syntax in raw strings
val price = """
${'$'}_9.99
"""
```



Arrays

Developer Student Clubs

Kotlin Arrays

Arrays in Kotlin are represented by the Array class. It has **get()** and **set()** functions that turn into **[]** by *operator overloading conventions*, and the **size** property, along with other useful member functions:

```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    operator fun iterator(): Iterator<T>
    // ...
}
```

Kotlin Arrays

To create an array, use the **function arrayOf()** and pass the item values to it, so that *arrayOf(1, 2, 3)* creates an array [1, 2, 3]. Alternatively, the **arrayOfNulls()** function can be used to create an array of a given size filled with null elements.

Another option is to use the **Array constructor** that takes the *array size and the function that returns values of array elements given its index*:

```
// Creates an Array<String> with values ["0", "1", "4", "9", "16"]
val asc = Array(5) { i -> (i * i).toString() }
asc.forEach { println(it) }
```

→ **The [] operation stands for calls to member functions get() and set().**

Kotlin Arrays

Arrays in Kotlin are ***invariant***.

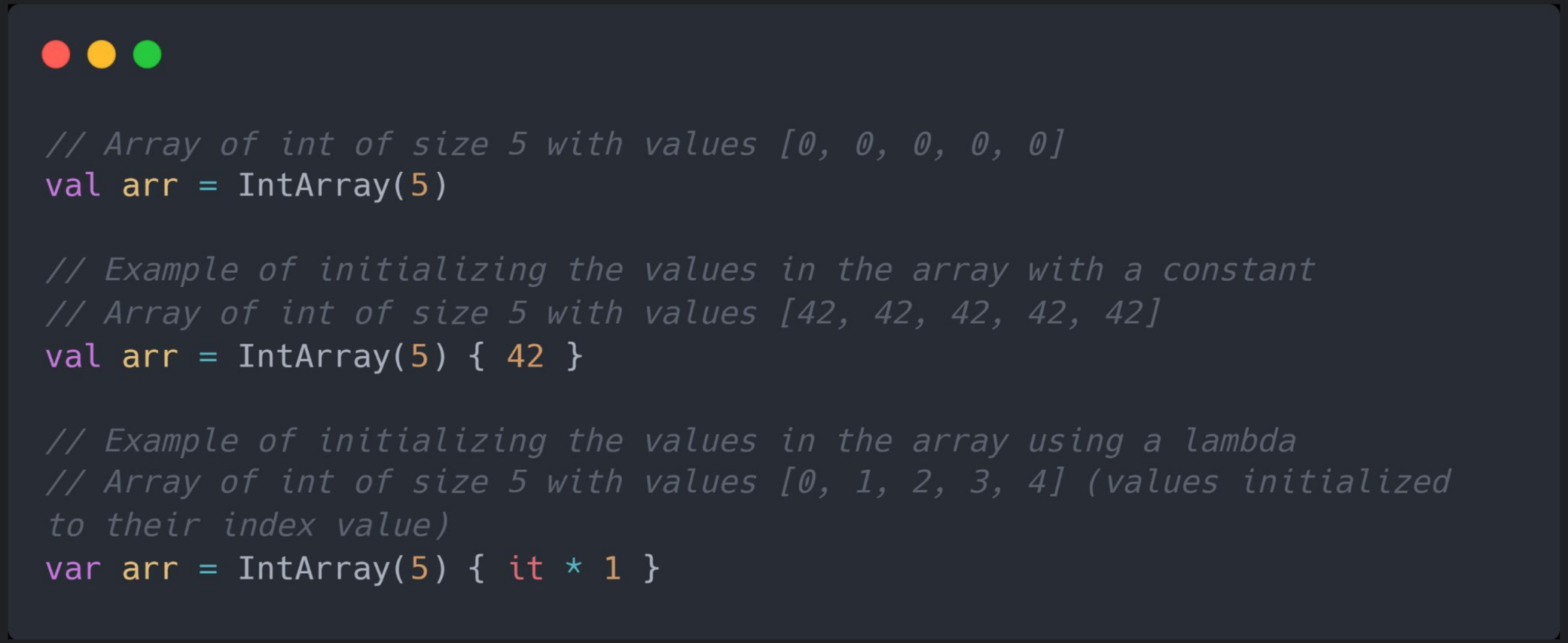
This means that Kotlin does not let us assign an *Array<String>* to an *Array<Any>*, which prevents a possible runtime failure (but you can use *Array<out Any>*, see Type Projections).

Primitive Type Arrays

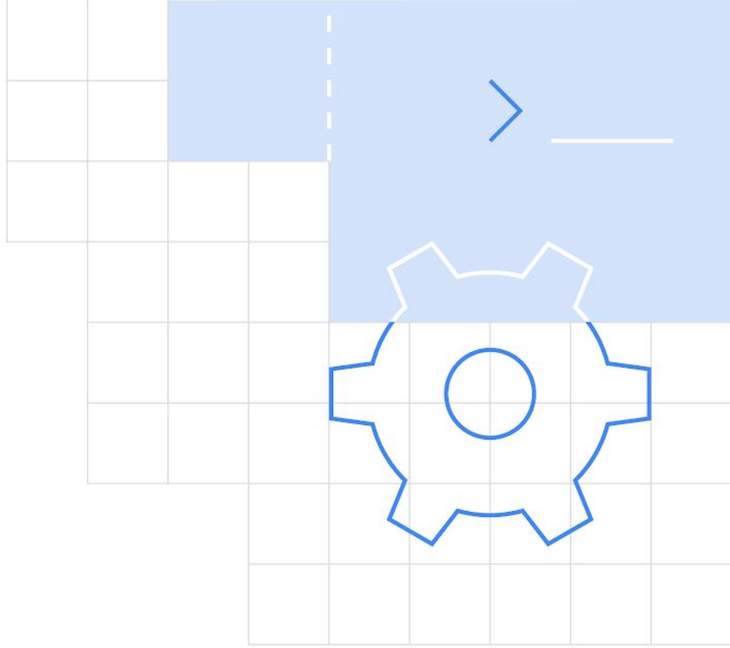
Kotlin also has classes that represent arrays of primitive types without boxing overhead: ByteArray, ShortArray, IntArray, and so on.

These classes have no inheritance relation to the Array class, but they have the same set of methods and properties. Each of them also has a corresponding factory function:

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```



```
// Array of int of size 5 with values [0, 0, 0, 0, 0]
val arr = IntArray(5)

// Example of initializing the values in the array with a constant
// Array of int of size 5 with values [42, 42, 42, 42, 42]
val arr = IntArray(5) { 42 }

// Example of initializing the values in the array using a lambda
// Array of int of size 5 with values [0, 1, 2, 3, 4] (values initialized
to their index value)
var arr = IntArray(5) { it * 1 }
```



Type Checks & Casting

Developer Student Clubs

The is & !is operators

Use the *is* operator or its negated form *!is* to perform a runtime check that identifies whether an object conforms to a given type:

```
if (obj is String) {
    print(obj.length)
}

if (obj !is String) { // same as !(obj is String)
    print("Not a String")
} else {
    print(obj.length)
}
```

- In most cases, you don't need to use explicit cast operators in Kotlin because the compiler tracks the is-checks and explicit casts for immutable values and inserts (safe) casts automatically when necessary:

```
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

- The compiler is smart enough to know that a cast is safe if a negative check leads to a return:

```
if (x !is String) return

print(x.length) // x is automatically cast to String
```

- or if it is on the right-hand side of && or || and the proper check (regular or negative) is on the left-hand side:

```
// x is automatically cast to String on the right-hand side of `||`
if (x !is String || x.length == 0) return

// x is automatically cast to String on the right-hand side of `&&`
if (x is String && x.length > 0) {
    print(x.length) // x is automatically cast to String
}
```

The as operator

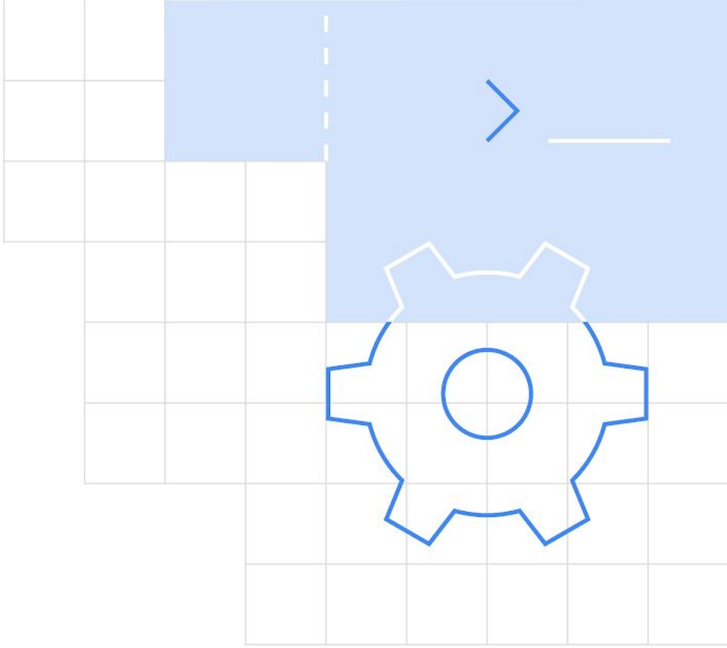
Usually, the cast operator throws an exception if the cast isn't possible. And so, it's called *unsafe*. The unsafe cast in Kotlin is done by the infix operator as.

```
val x: String = y as String
```

Note that null cannot be cast to String, as this type is not nullable. If y is null, the code above throws an exception. To make code like this correct for null values, use the nullable type on the right-hand side of the cast:

```
val x: String? = y as String?
```

Functions



Developer Student Clubs

Kotlin Functions

Kotlin's functions are declared using the fun keyword! For example:

```
fun double(x: Int): Int {
    return 2 * x
}
```

They are called using the standard approach, i.e. *double(5)* while member function calls use the dot notation. For example: *Stream().read()*

Function Parameters

Function parameters are defined using **Pascal notation** - *name*: *type*. Parameters are separated using commas, and each parameter must be explicitly typed.

Function parameters can have default values, which are used when you skip the corresponding argument. This reduces the number of overloads:

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

Unit Returning Functions

If a function does not return a useful value, its return type is Unit. Unit is a type with only one value - Unit. This value does not have to be returned explicitly:

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

The Unit return type declaration is also optional. The above code is equivalent to:

```
fun printHello(name: String?) { ... }
```

Single Expression Functions

When a function returns a single expression, the curly braces can be omitted and the body is specified after a = symbol:

```
fun double(x: Int): Int = x * 2
```

Explicitly declaring the return type is optional when this can be inferred by the compiler:

```
fun double(x: Int) = x * 2
```

Variable number of arguments

You can mark a parameter of a function (usually the last one) with the vararg modifier:

```
fun <T> asList(vararg ts: T): List<T> {
   val result = ArrayList<T>()
   for (t in ts) // ts is an Array
      result.add(t)
   return result
  }
```

In this case, you can pass a variable number of arguments to the function:

```
  val list = asList(1, 2, 3)
```

Inside a function, a vararg-parameter of type T is visible as an array of T, as in the example above, where the ts variable has type Array<out T>.

Only one parameter can be marked as vararg. If a vararg parameter is not the last one in the list, values for the subsequent parameters can be passed using named argument syntax, or, if the parameter has a function type, by passing a lambda outside the parentheses.

Function Scope

Kotlin functions can be declared at the top level in a file, meaning **you do not need to create a class to hold a function**, which you are required to do in languages such as Java, C#, and Scala (top level definition is available since Scala 3). In addition to top level functions, Kotlin functions can also be declared locally as member functions and extension functions.

Local functions

Kotlin supports local functions, which are *functions inside other functions*:

```
fun dfs(graph: Graph) {
  fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
    if (!visited.add(current)) return
    for (v in current.neighbors)
      dfs(v, visited)
  }

  dfs(graph.vertices[0], HashSet())
}
```

Function Scope

A local function can access local variables of outer functions (the closure). In the case above, visited can be a local variable:

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

Function Scope

Member functions

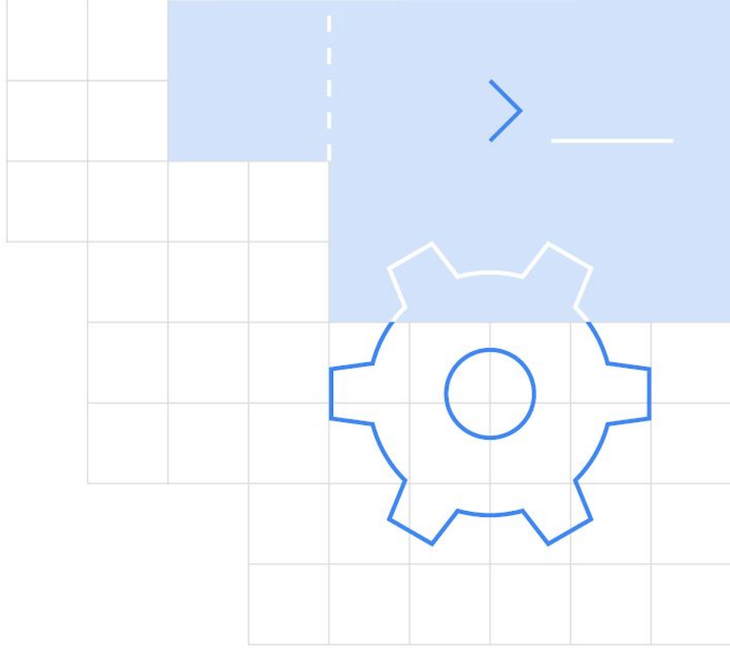
A member function is a function that is defined inside a class or object:

```
class Sample {
    fun foo() { print("Foo") }
}
```

Member functions are called with dot notation like we mentioned earlier:

```
Sample().foo() // creates instance of class Sample and calls foo
```

Control Flow



Developer Student Clubs

If Expression

In Kotlin, **if is an expression**: it returns a value. Therefore, there is no ternary operator (condition ? then : else) because ordinary if works fine in this role. Example:

```
var max = a
if (a < b) max = b

// With else
var max: Int
if (a > b) {
    max = a
} else {
    max = b
}

// As expression
  val max = if (a > b) a else b
```

Branches of an if expression can be blocks. In this case, the last expression is the value of a block:

```
val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
  }
```

If you're using if as an expression, for example, for returning its value or assigning it to a variable, the else branch is mandatory.

The When Expression

when defines a conditional expression with multiple branches. It is similar to the switch statement in C-like languages. Its simple form looks like this.

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

when matches its argument against all branches sequentially until some branch condition is satisfied. when can be used either as an expression or as a statement. If it is used as an expression, the value of the first matching branch becomes the value of the overall expression. If it is used as a statement, the values of individual branches are ignored. Just like with if, each branch can be a block, and its value is the value of the last expression in the block. The else branch is evaluated if none of the other branch conditions are satisfied.

The When Expression

If when is used as an *expression*, the else branch is mandatory, unless the compiler can prove that all possible cases are covered with branch conditions, for example, with enum class entries and sealed class subtypes.

```
enum class Bit {
  ZERO, ONE
}


val numericValue = when (getRandomBit()) {
    Bit.ZERO -> 0
    Bit.ONE -> 1
    // 'else' is not required because all cases are covered
}
```

For Loops

The for loop iterates through anything that provides an iterator. This is equivalent to the foreach loop in languages like C#. The syntax of for is the following:

```
for (item in collection) print(item)
```

The body of for can be a block.

```
for (item: Int in ints) {
    // ...
}
```

We will see more example of the for loop later on, when we will cover ranges!

While Loops

While and do-while loops execute their body continuously while their condition is satisfied. The difference between them is the condition checking time:

- while checks the condition and, if it's satisfied, executes the body and then returns to the condition check.
- do-while executes the body and then checks the condition. If it's satisfied, the loop repeats. So, the body of do-while executes at least once regardless of the condition.

```
while (x > 0) {
    x--
}


do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

Return & Jumps

Kotlin has three structural jump expressions:

- return by default returns from the nearest enclosing function or anonymous function.
- break terminates the nearest enclosing loop.
- continue proceeds to the next step of the nearest enclosing loop.

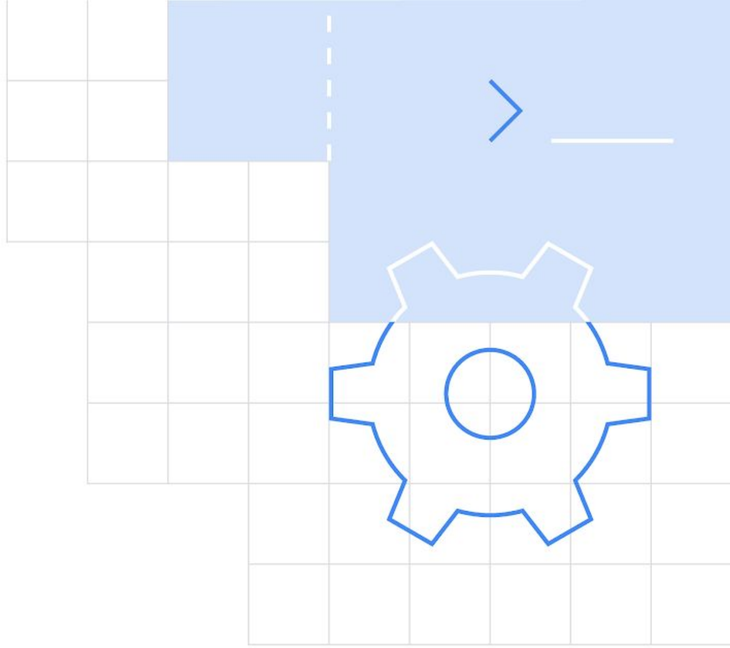
All of these expressions can be used as part of larger expressions:

```
val s = person.name ?: return
```

The type of these expressions is the Nothing type.

Kotlin also supports labels for break, continue & return statements! Find more here.

Classes & Objects



Developer Student Clubs

Classes

Classes in Kotlin are declared using the keyword class:

```
class Person { /*...*/ }
```

The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor, and some other things), and the class body surrounded by curly braces. Both the header and the body are optional; if the class has no body, the curly braces can be omitted.

```
class Empty
```

Classes

A class in Kotlin can have a *primary constructor* and one or more *secondary constructors*. The primary constructor is a part of the class header, and it goes after the class name and optional type parameters.

```
class Person constructor(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted:

```
class Person(firstName: String) { /*...*/ }
```

The primary constructor cannot contain any code. Initialization code can be placed in *initializer blocks* prefixed with the init keyword.

Classes

During the initialization of an instance, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
```

Classes

Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

Such declarations can also include default values of the class properties:

```
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = true)
```

Much like regular properties, properties declared in the primary constructor can be mutable (var) or read-only (val).

If the constructor has annotations or visibility modifiers, the constructor keyword is required and the modifiers go before it:

```
class Customer public @Inject constructor(name: String) { /*...*/ }
```

Instantiating Classes

To create an instance of a class, call the constructor as if it were a regular function:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

Kotlin does not have a new keyword.

The process of creating instances of nested, inner, and anonymous inner classes is described in Nested classes.

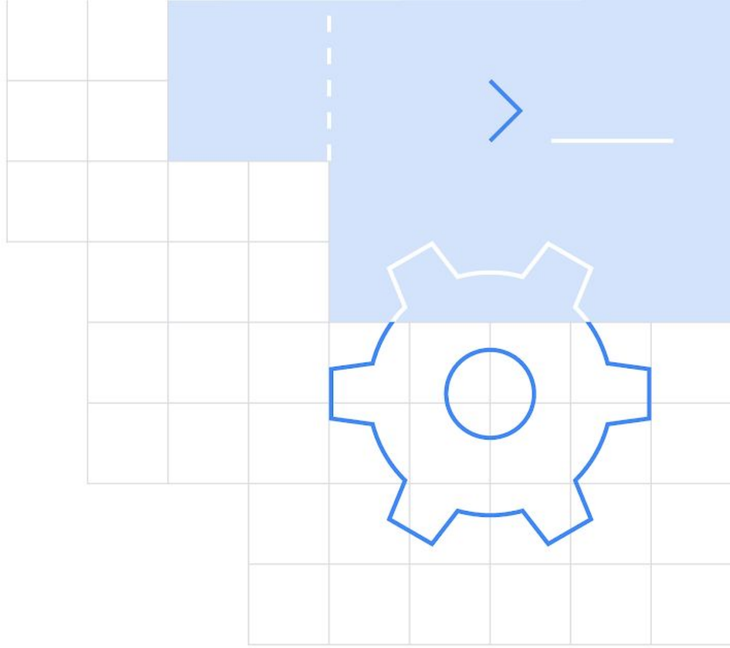
Class Members

Classes can contain:

- Constructors and initializer blocks
- Functions
- Properties
- Nested and inner classes
- Object declarations

There is a myriad of things left to cover about classes, such as Inheritance, Abstract classes, Interfaces, Sealed classes, In/Out Classes & much more! Some of those will be covered in our next event, but you are strongly encouraged to skim through the documentation!

Kotlin Ranges



Developer Student Clubs

Ranges

Kotlin lets you easily create ranges of values using the rangeTo() function from the kotlin.ranges package and its operator form ... Usually, rangeTo() is complemented by in or !in functions.

```
if (i in 1..4) { // equivalent of i >= 1 && i <= 4
    print(i)
}
```

Integral type ranges (IntRange, LongRange, CharRange) have an extra feature: they can be iterated over. These ranges are also progressions of the corresponding integral types.

Range definition

A range defines a closed interval in the mathematical sense: **it is defined by its two endpoint values which are both included in the range**. Ranges are defined for comparable types: having an order, you can define whether an arbitrary instance is in the range between two given instances.

The main operation on ranges is **contains**, which is usually used in the form of in and !in operators.

To create a range for your class, call the *rangeTo()* function on the range start value and provide the end value as an argument. *rangeTo()* is often called in its operator form ...

```
val versionRange = Version(1, 11)..Version(1, 30)
println(Version(0, 9) in versionRange)
println(Version(1, 20) in versionRange)
```

Iterating using ranges

Ranges are generally used for iteration in for loops.

```
for (i in 1..4) print(i)
```

To iterate numbers in reverse order, use the downTo function instead of the... operator:

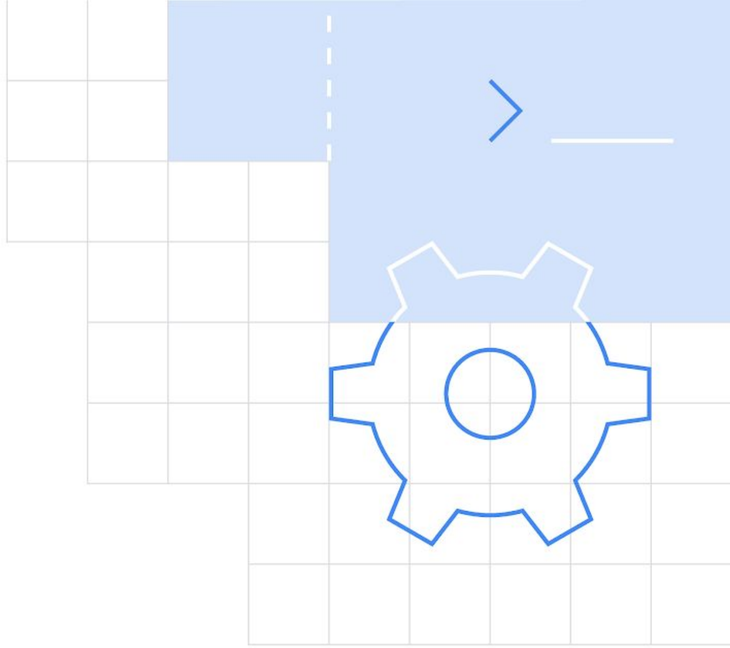
```
for (i in 4 downTo 1) print(i)
```

It is also possible to iterate over numbers with an arbitrary step (not necessarily 1). This is done via the step function:

```
for (i in 1..8 step 2) print(i)

println()

for (i in 8 downTo 1 step 2) print(i)
```

To iterate a number range which does not include its end element, use the until function:

```
for (i in 1 until 10) {      // i in 1 until 10, excluding 10
  print(i)
}
```



Kotlin Collections

Developer Student Clubs

Collections Overview

The Kotlin Standard Library provides a comprehensive set of tools for managing *collections* – groups of a variable number of items (possibly zero) that are significant to the problem being solved and are commonly operated on.

It also provides implementations for basic collection types: sets, lists, and maps. A pair of interfaces represent each collection type:

- A *read-only* interface that provides operations for accessing collection elements.
- A *mutable* interface that extends the corresponding read-only interface with write operations: adding, removing, and updating its elements.

List<T> Collection

- List<T> stores elements in a specified order and provides indexed access to them. Indices start from zero – the index of the first element – and go to lastIndex which is the (list.size - 1). For example: *val numbers = listOf("one", "two", "three", "four")*
- MutableList<T> is a List with list-specific write operations, for example, to add or remove an element at a specific position.

```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

As you see, in some aspects, lists are very similar to arrays. However, there is **one important difference**: an array's size is defined upon initialization and is never changed; in turn, a list doesn't have a predefined size; a list's size can be changed as a result of write operations: adding, updating, or removing elements.

In Kotlin, the default implementation of List is ArrayList which you can think of as a resizable array.

Set<T> Collection

Set<T> stores unique elements; their order is generally undefined. null elements are unique as well: a Set can contain only one null. Two sets are equal if they have the same size, and for each element of a set there is an equal element in the other set.

```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")
if (numbers.contains(1)) println("1 is in the set")

val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal: ${numbers == numbersBackwards}")
```

MutableSet<T> Collection

MutableSet is a Set with write operations from MutableCollection.
The default implementation of Set – LinkedHashSet – preserves the order of elements insertion. Hence, the functions that rely on the order, such as first() or last(), return predictable results on such sets.

```
val numbers = setOf(1, 2, 3, 4)  // LinkedHashSet is the default implementation
val numbersBackwards = setOf(4, 3, 2, 1)
println(numbers.first() == numbersBackwards.first())
println(numbers.first() == numbersBackwards.last())
```

An alternative implementation – HashSet – says nothing about the elements order, so calling such functions on it returns unpredictable results. However, HashSet requires less memory to store the same number of elements.

Map<K,V> Collection

Map<K, V> is not an inheritor of the Collection interface; however, it's a Kotlin collection type as well. A Map stores *key-value* pairs (or *entries*); keys are unique, but different keys can be paired with equal values. The Map interface provides specific functions, such as access to value by key, searching keys and values, and so on.

```
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}")
if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
if (1 in numbersMap.values) println("The value 1 is in the map")
if (numbersMap.containsValue(1)) println("The value 1 is in the map") // same as previous
```

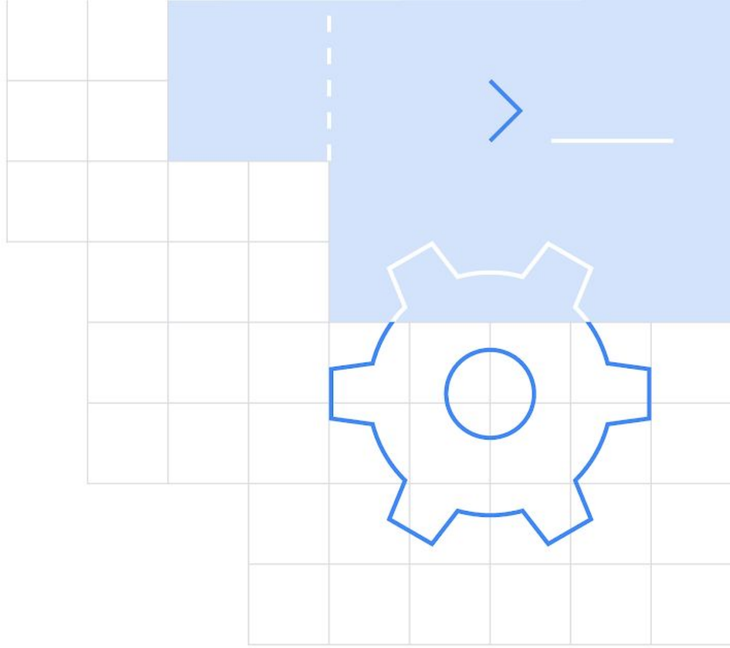
MutableMap<K,V> Collection

MutableMap is a Map with map write operations, for example, you can add a new key-value pair or update the value associated with the given key.

```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
numbersMap["one"] = 11
println(numbersMap)
```

The default implementation of Map – LinkedHashMap – preserves the order of elements insertion when iterating the map. In turn, an alternative implementation – HashMap – says nothing about the elements order.

Null Safety!!



Developer Student Clubs

Nullable & non-nullable types

Kotlin's type system is aimed at **eliminating the danger of null references**, also known as The Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java this would be the equivalent of a NullPointerException, or an *NPE* for short.
The only possible causes of an NPE in Kotlin are:

- An explicit call to throw NullPointerException().
- Usage of the !! operator that is described later.
- Data inconsistency with regard to initialization, such as when:
  - An uninitialized this available in a constructor is passed and used somewhere (a "leaking this").
  - A superclass constructor calls an open member whose implementation in the derived class uses an uninitialized state.
- Java interoperation:
  - Attempts to access a member of a null reference of a platform type;
  - Nullability issues with generic types being used for Java interoperation. For example, a piece of Java code might add null into a Kotlin MutableList<String>, therefore requiring a MutableList<String?> for working with it.
  - Other issues caused by external Java code.

Nullable & non-nullable types

In Kotlin, the type system distinguishes between references that can hold null (**nullable** references) and those that cannot (**non-null** references). For example, a regular variable of type String cannot hold null:

```
var a: String = "abc" // Regular initialization means non-null by default

a = null // compilation error
```

To allow nulls, you can declare a variable as a nullable string by writing String?:

```
var b: String? = "abc" // can be set to null

b = null // ok

print(b)
```

Now, if you call a method or access a property on a, it's guaranteed not to cause an NPE, so you can safely say:

```
val l = a.length
```

But if you want to access the same property on b, that would not be safe, and the compiler reports an error:

```
val l = b.length // error: variable 'b' can be null
```

But you still need to access that property, right? Let’s see how to achieve this!

Checking for null conditions

First, you can explicitly check whether b is null, and handle the two options separately:

```
val l = if (b != null) b.length else -1
```

The *compiler tracks the information* about the check you performed, and allows the call to length inside the if. More complex conditions are supported as well:

```
val b: String? = "Kotlin"
if (b != null && b.length > 0) {
    print("String of length ${b.length}")
} else {
    print("Empty string")
}
```

Note that **this only works where b is immutable** (meaning it is a local variable that is not modified between the check and its usage or it is a member val that has a backing field and is not overridable), because otherwise it could be the case that b changes to null after the check.

Safe calls

Your second option for accessing a property on a nullable variable is using the safe call operator ?. :

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length) // Unnecessary safe call
```

This returns b.length if b is not null, and null otherwise. The type of this expression is Int?.

Safe calls are useful in chains. For example, Bob is an employee who may be assigned to a department (or not). That department may in turn have another employee as a department head. To obtain the name of Bob's department head (if there is one), you write the following:

```
bob?.department?.head?.name
```

Such a chain returns null if **any of the properties** in it is null.

Safe calls

To perform a certain operation **only for non-null values**, you can use the safe call operator together with let:

```
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
  item?.let { println(it) } // prints Kotlin and ignores null
}
```

A safe call can also be placed on the left side of an assignment. Then, if one of the receivers in the safe calls chain is null, the assignment is skipped and the expression on the right is not evaluated at all:

```
// If either `person` or `person.department` is null, the function is not called:
person?.department?.head = managersPool.getManager()
```

The elvis operator

When you have a nullable reference, b, you can say "if b is not null, use it, otherwise use some non-null value":

```
val l: Int = if (b != null) b.length else -1
```

Instead of writing the complete if expression, you can also express this with the Elvis operator ?: :

```
val l = b?.length ?: -1
```

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.

Since throw and return are expressions in Kotlin, they can also be used on the right-hand side of the Elvis operator. This can be handy, for example, when checking function arguments:

```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

The !! operator

The third option is for NPE-lovers: the not-null assertion operator (!!) converts any value to a non-null type and throws an exception if the value is null. You can write b!!, and this will return a non-null value of b (for example, a String in our example) or throw an NPE if b is null:

```
val l = b!!.length
```

Thus, if you want an NPE, you can have it, but you have to ask for it explicitly and it won't appear out of the blue.

Don’t say we didn’t warn you ;)

Safe Casts

Regular casts may result in a ClassCastException if the object is not of the target type. Another option is to use safe casts that return null if the attempt was not successful:

```
val aInt: Int? = a as? Int
```

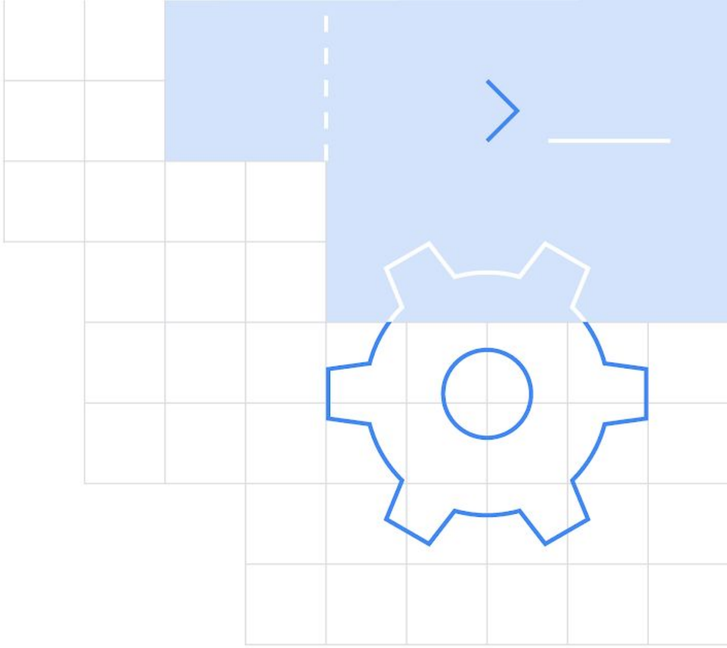
Closing Words

Thank you for your time everybody!

We hope that you learned something new today. There's a TON of awesome things that we couldn't cover today, so if you're interested in diving a bit deeper, here are a few resources:

- Kotlin Docs
- Kotlin Playground
- Kotlin by example
- Kotlin Hands-On

Special thanks to the @GoogleDevs that are making initiatives such as this possible! Don't forget to **follow the GDSC UoC on our social media & join our discord server** to stay up to date with upcoming events!



Thank you for your time!

Developer Student Clubs