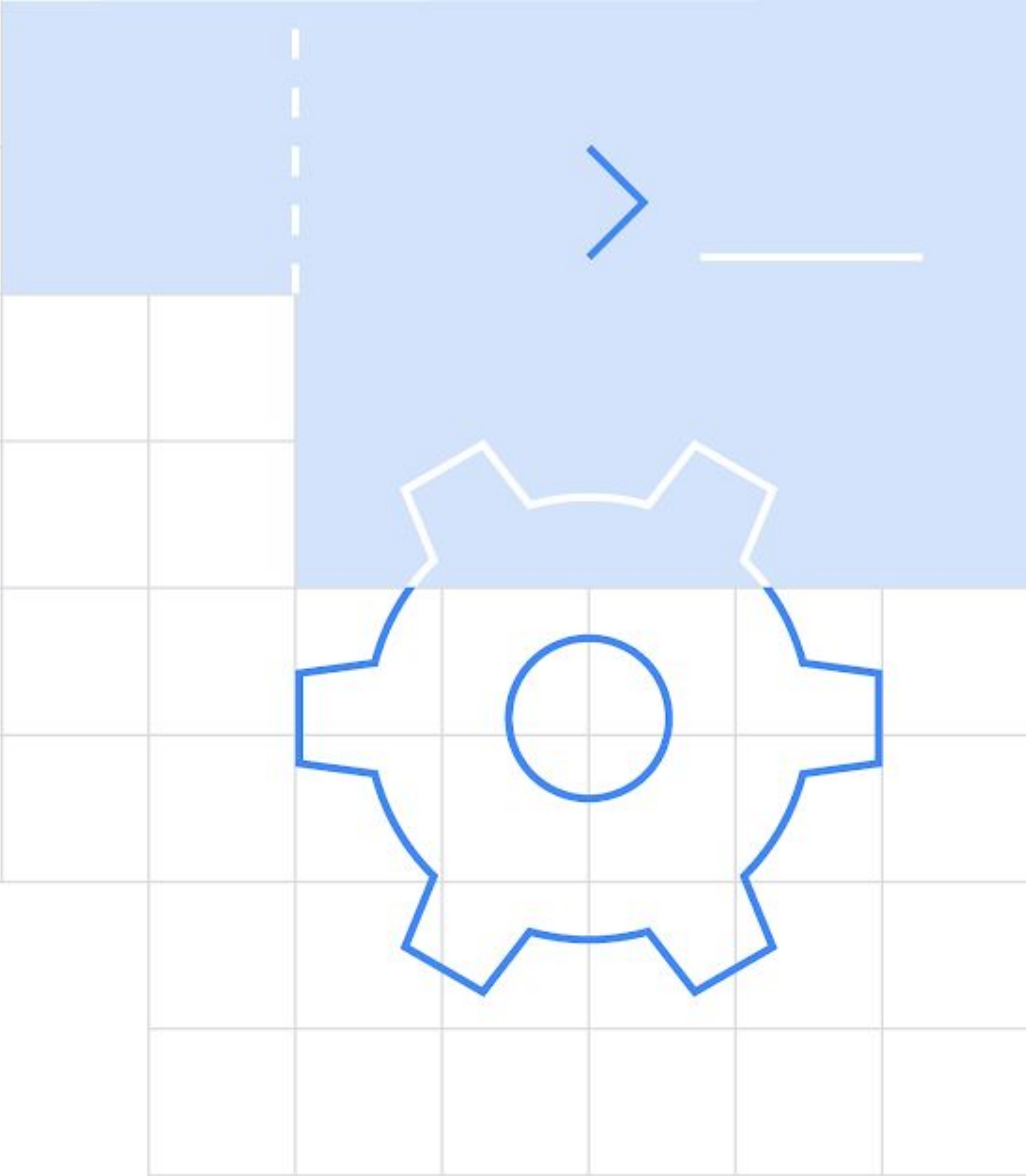
Developer Student Clubs

Kotlin Basics

An introductory course to the Kotlin programming language

Stelios Papamichail
Android Engineer
@MikePapamichail

Google Developers



Agenda

- Basic Syntax
- Types
- Variables
- Functions

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

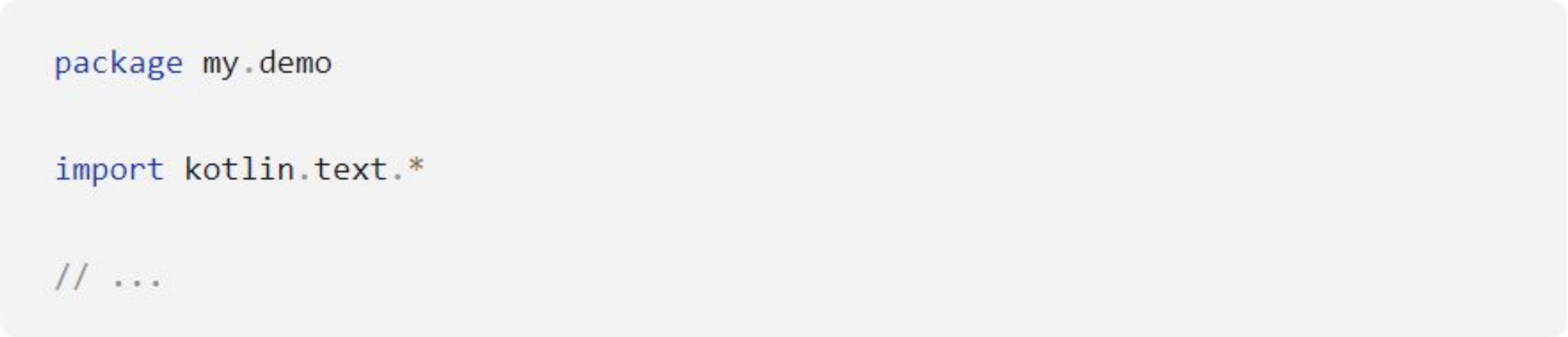
Basic Syntax

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.



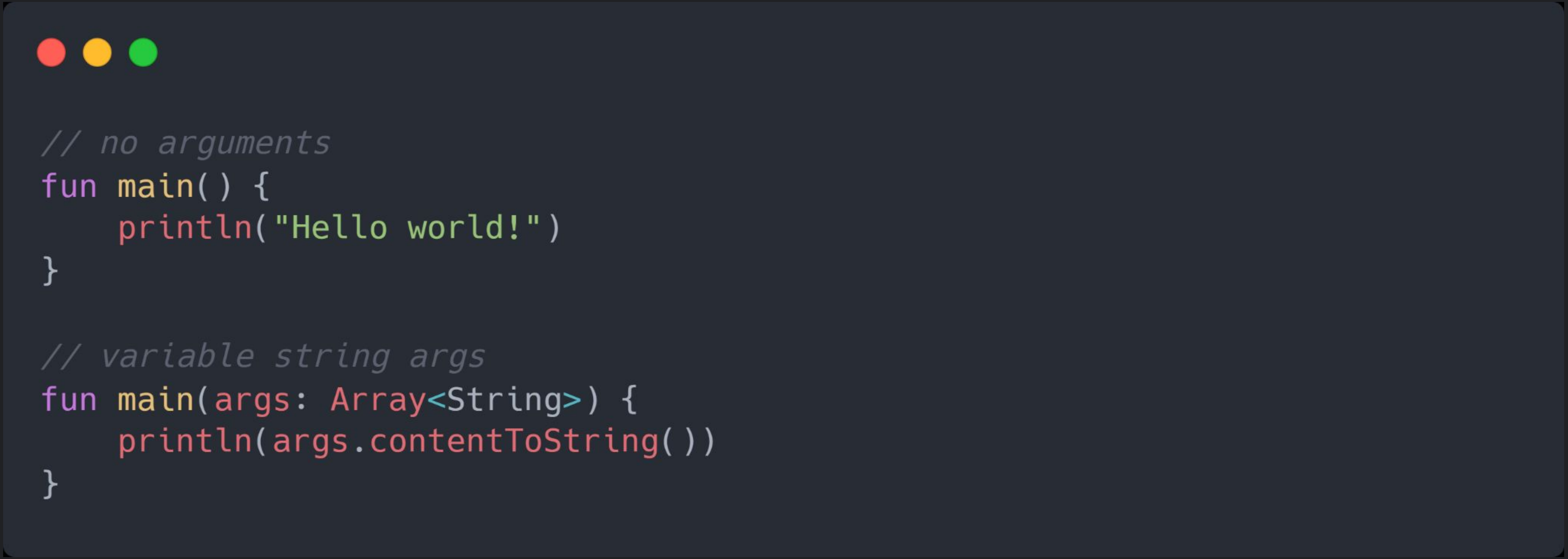
```
package my.demo

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 usualy 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.

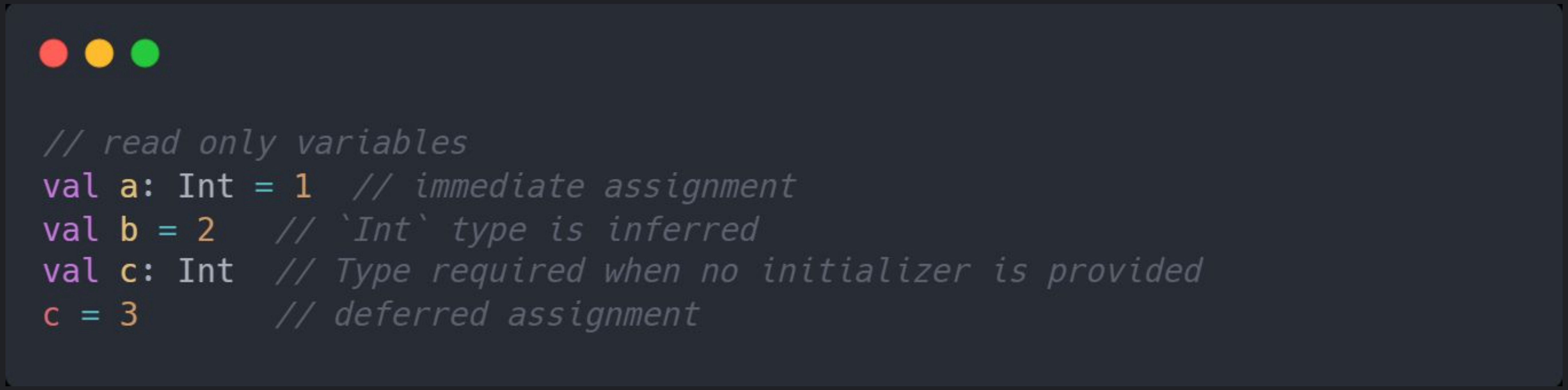


```kotlin
// no arguments
fun main() {
    println("Hello world!")
}

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

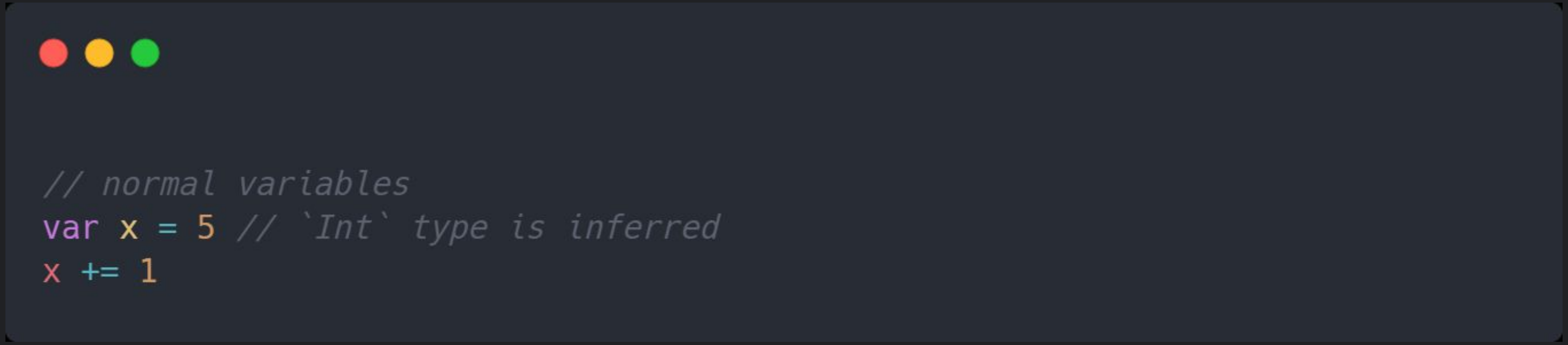
Variables

Read-only, read/write & global variables

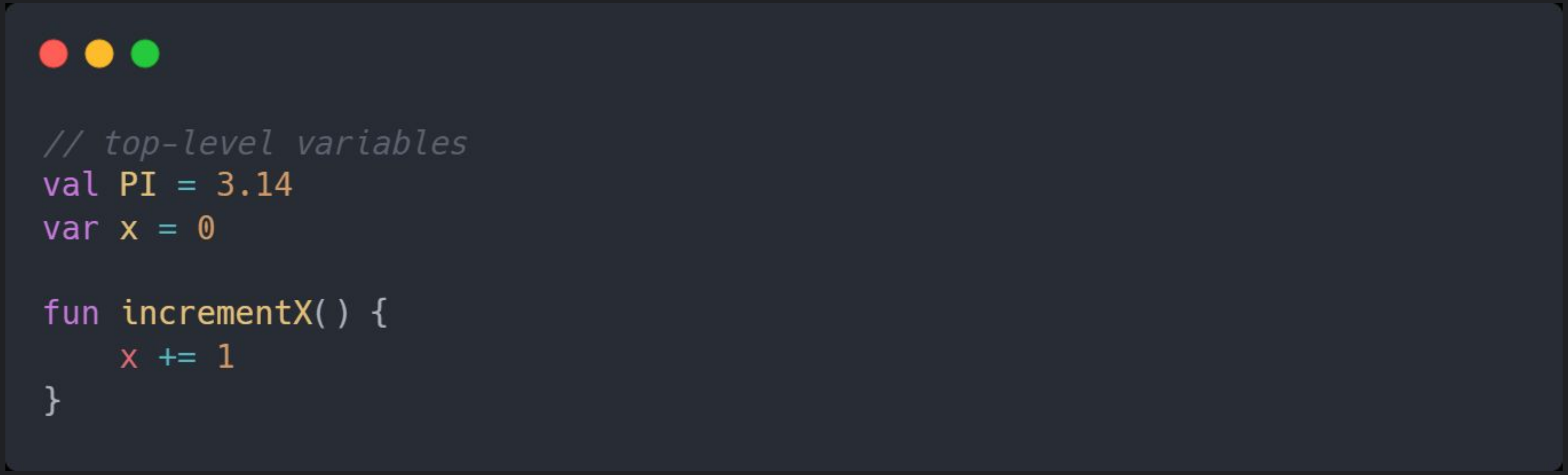
- **Read-only** local variables are defined using the keyword val. They can be assigned a value only once.
- Variables that can be **reassigned** use the var keyword.
- You can declare variables at the top level.



```kotlin
// 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
```



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



```kotlin
// 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.

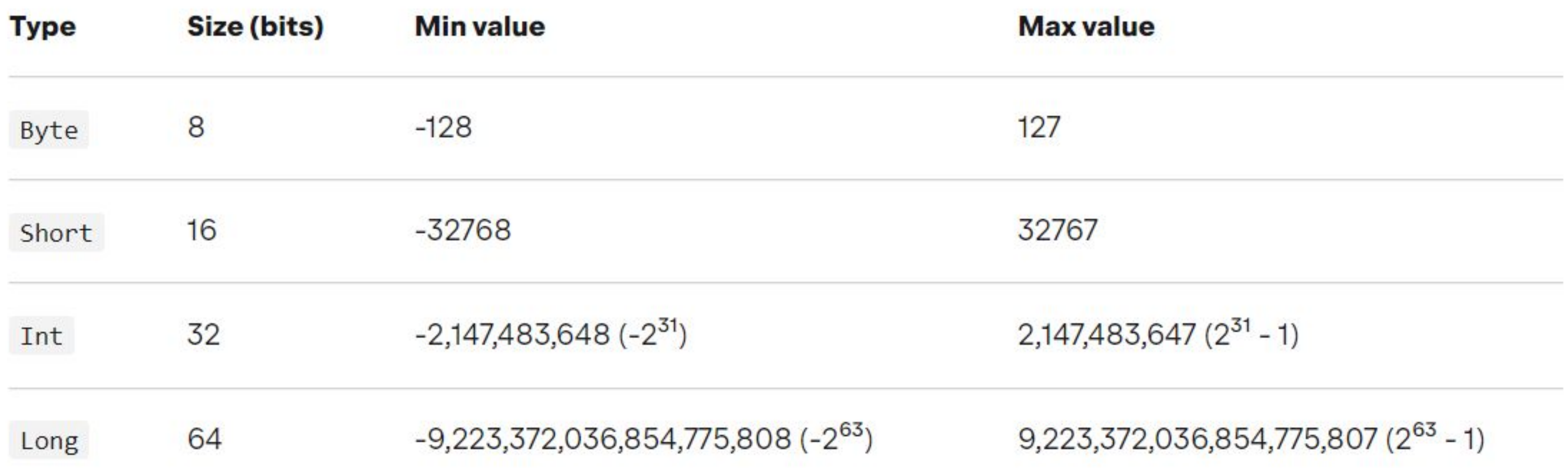
We will briefly cover the following basic types: **Numbers** & their unsigned counterparts, **Booleans**, **Characters**, **Strings** & **Arrays**!

Signed Integer Types

Developer Student Clubs

Integer Types

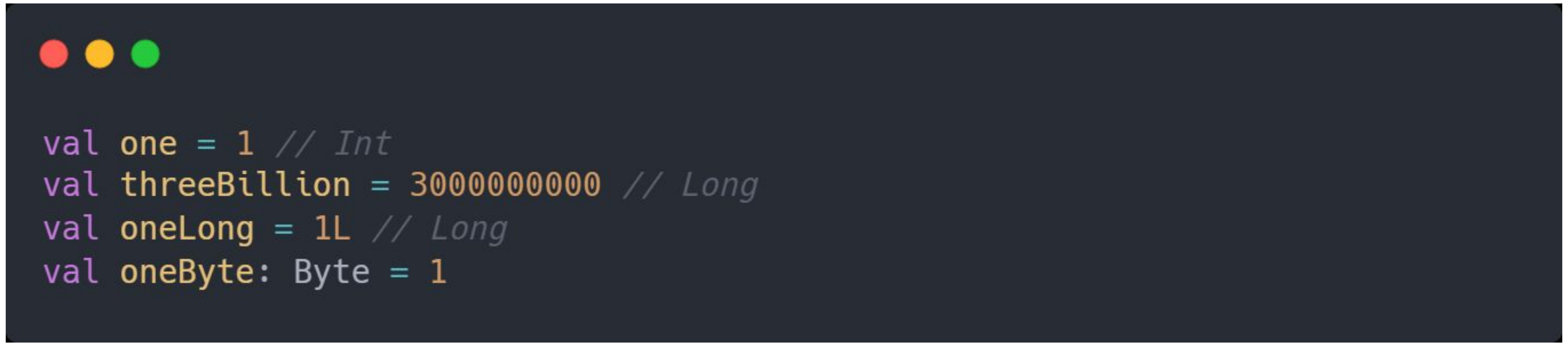
Kotlin provides a set of built-in types that represent numbers. For integer numbers, there are four types with different sizes and, hence, value ranges:



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.



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

Unsigned Integers

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:

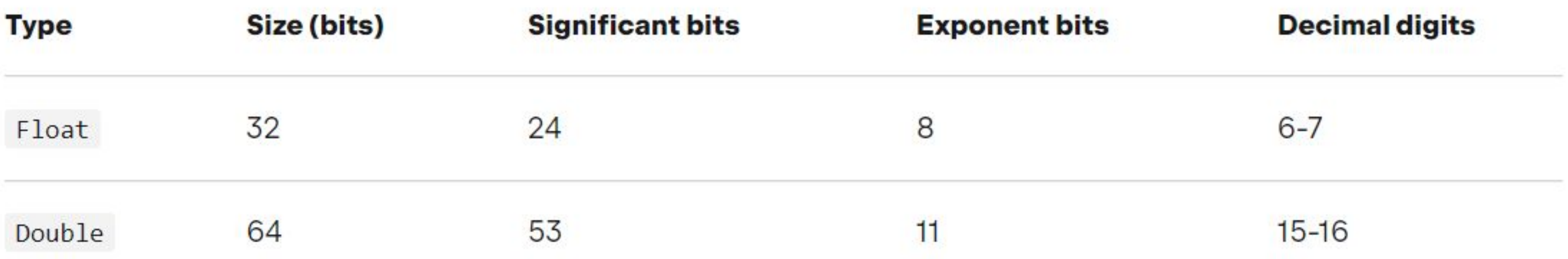
- Using signed integers can help to detect accidental overflows and signal error conditions, such as List.lastIndex being -1 for an empty list.
- Unsigned integers cannot be treated as a range-limited version of signed ones because their range of values is not a subset of the signed integers range. Neither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

Floating Point Types

Developer Student Clubs

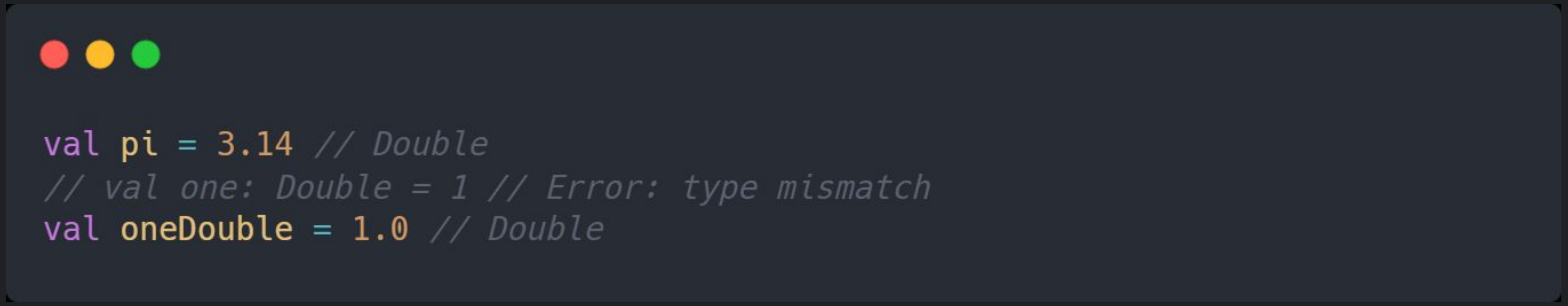
Floating Point Types

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:



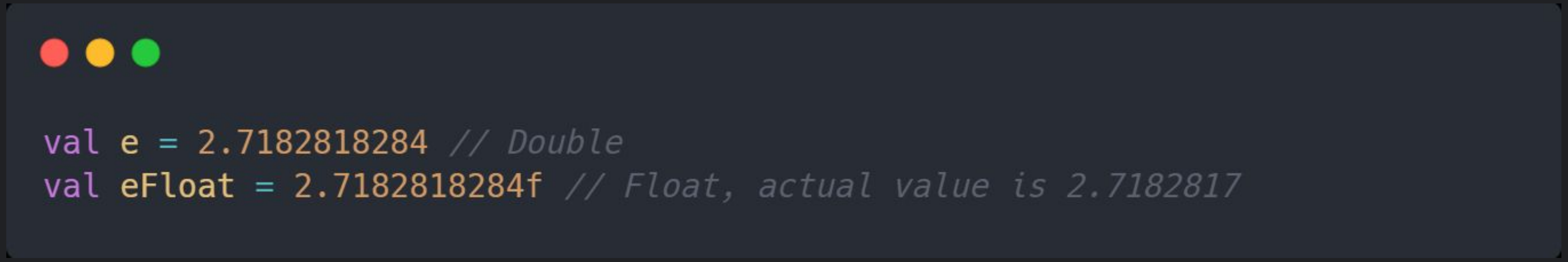
Type	Size (bits)	Significant bits	Exponent bits	Decimal digits
`Float`	32	24	8	6-7
`Double`	64	53	11	15-16

- 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:



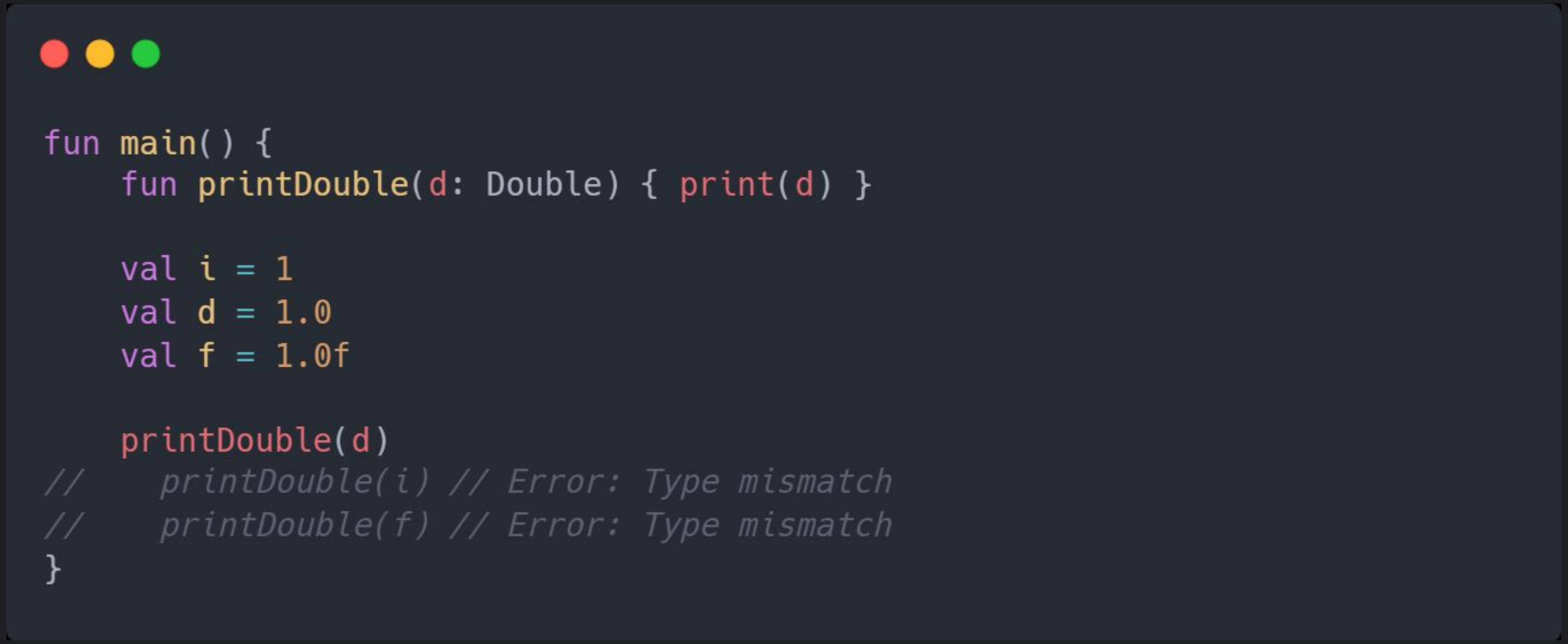
```kotlin
val pi = 3.14 // Double
// val one: Double = 1 // Error: type mismatch
val oneDouble = 1.0 // Double
```

- 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:



```kotlin
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:



```kotlin
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:

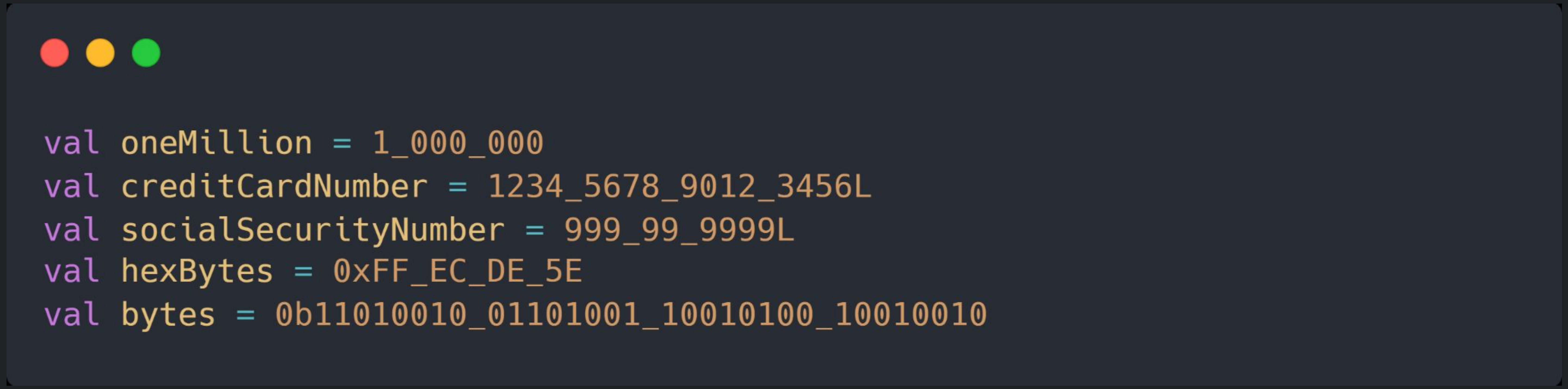
- **Decimals**: 123
    - Longs are tagged by a capital L: 123L
- **Hexadecimals**: 0x0F
- **Binaries**: 0b00001011

⟶ Octal literals are not supported in Kotlin.

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

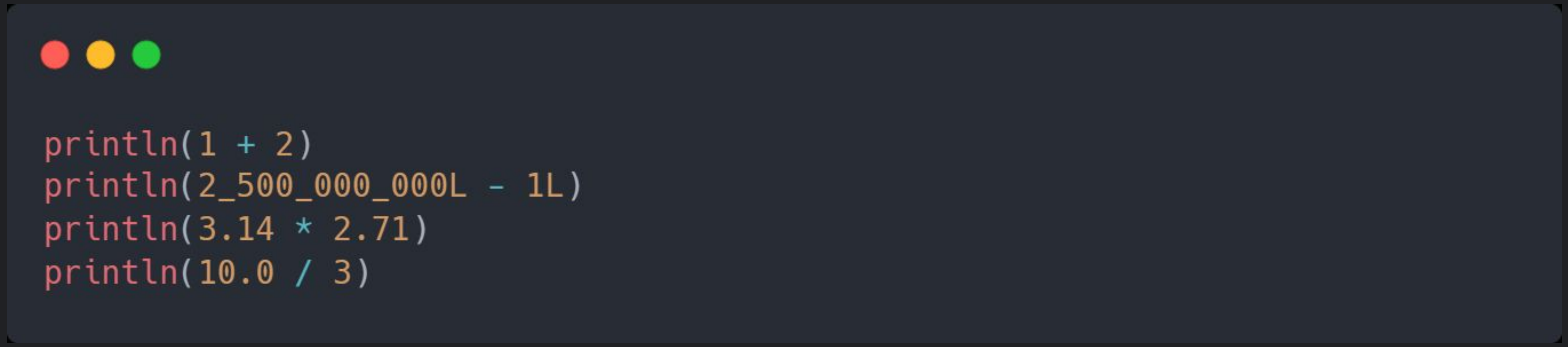
- **Doubles** by default: 123.5, 123.5e10
- **Floats** are tagged by f or F: 123.5f

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



```kotlin
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:



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

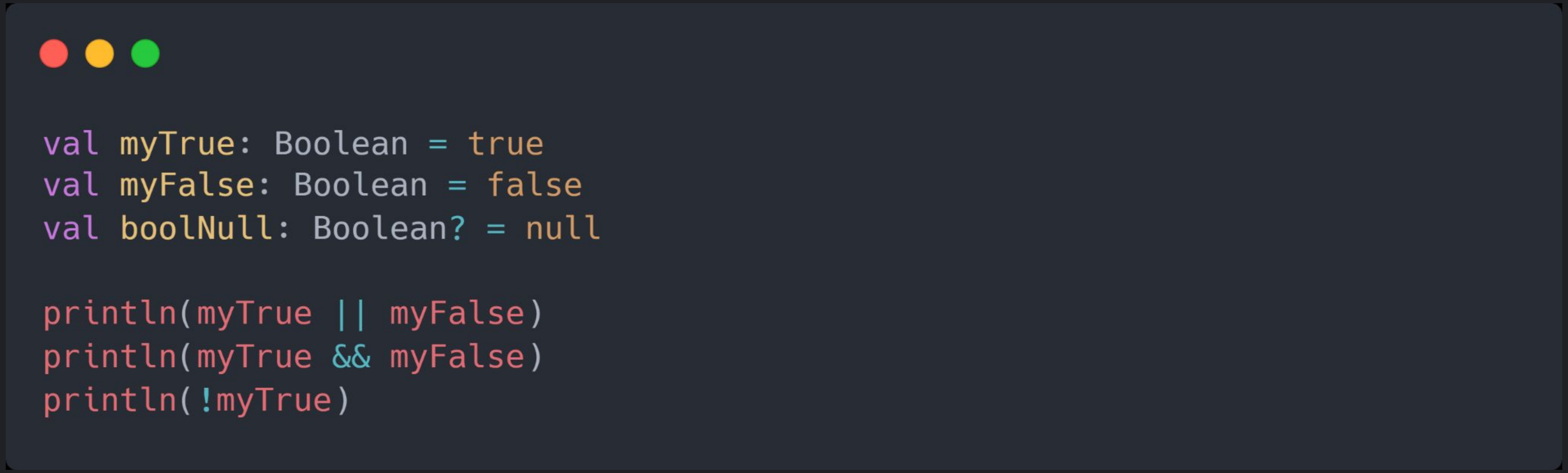
The Boolean type

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

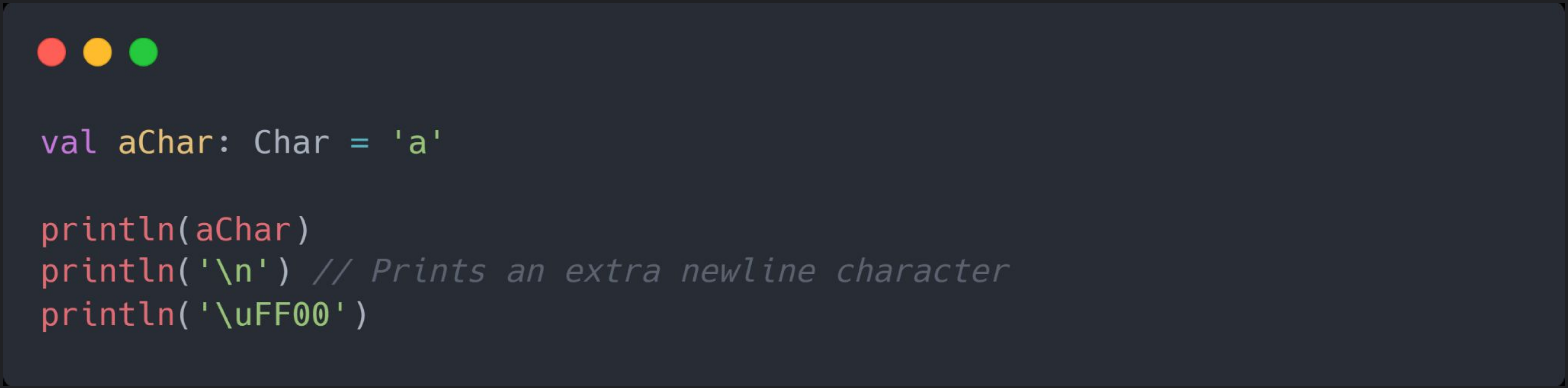
Developer Student Clubs

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'.



```kotlin
val aChar: Char = 'a'

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

Strings & String literals

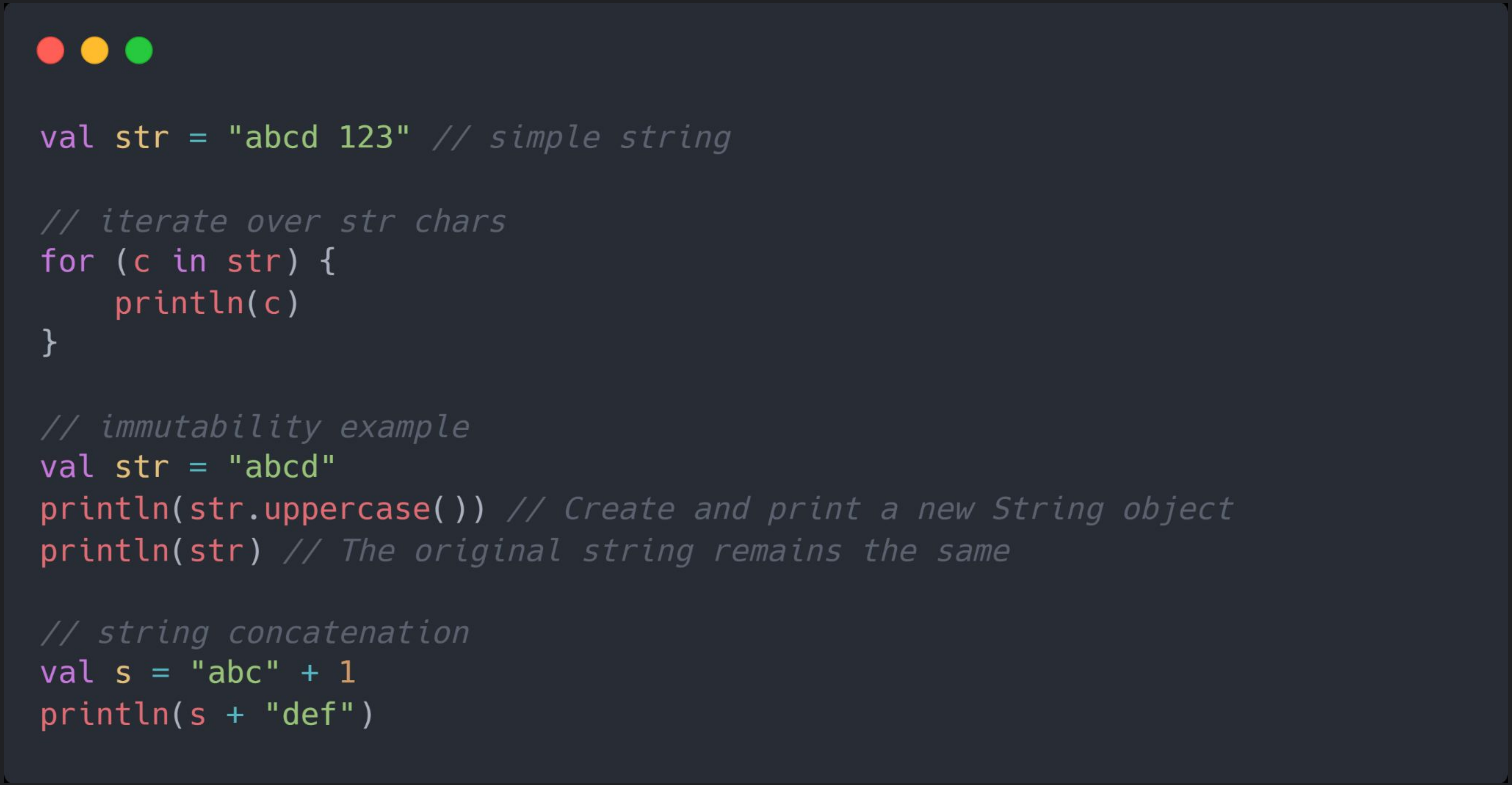
String Type

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

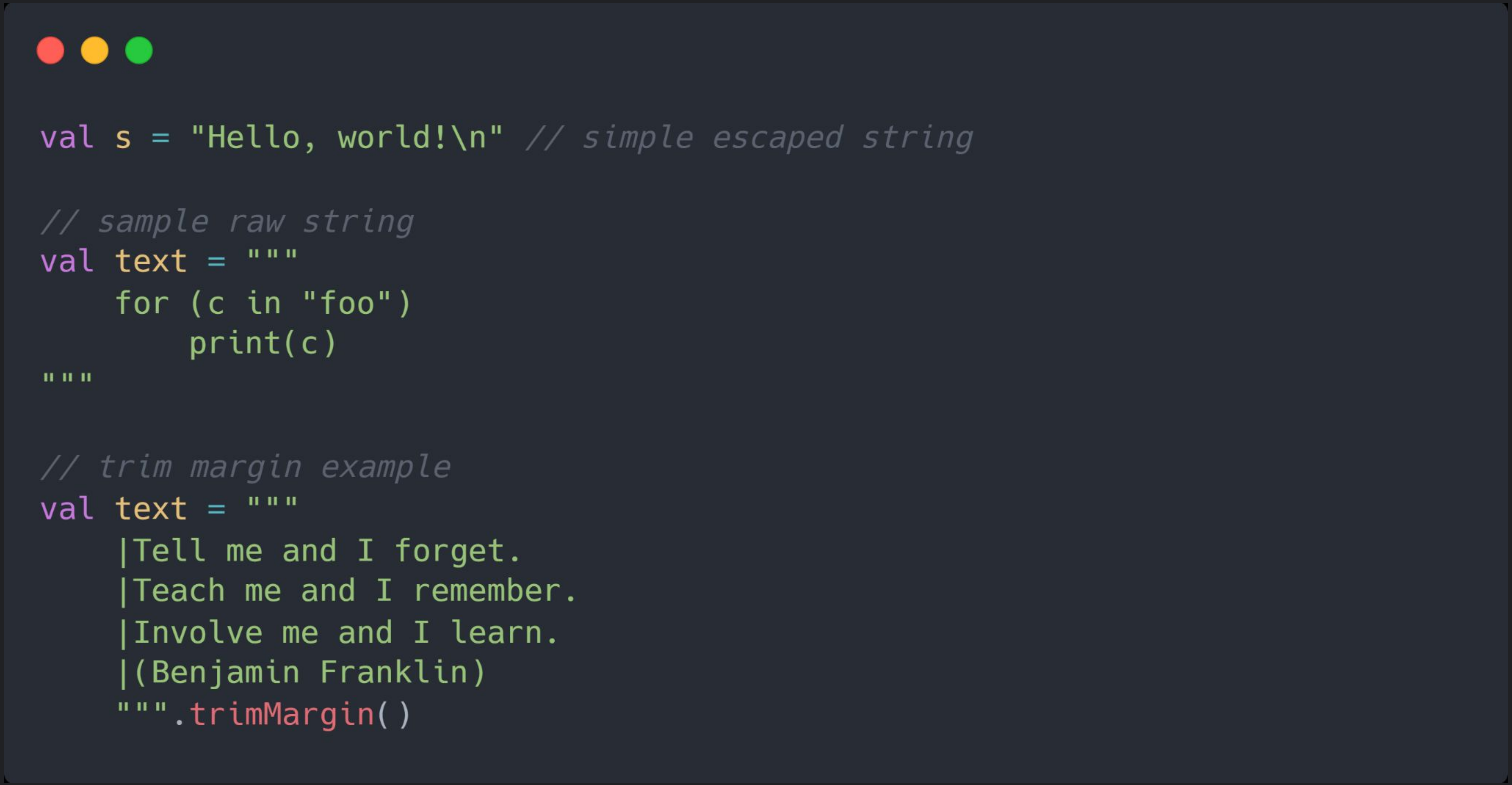
Kotlin has two types of string literals:

- 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()
```

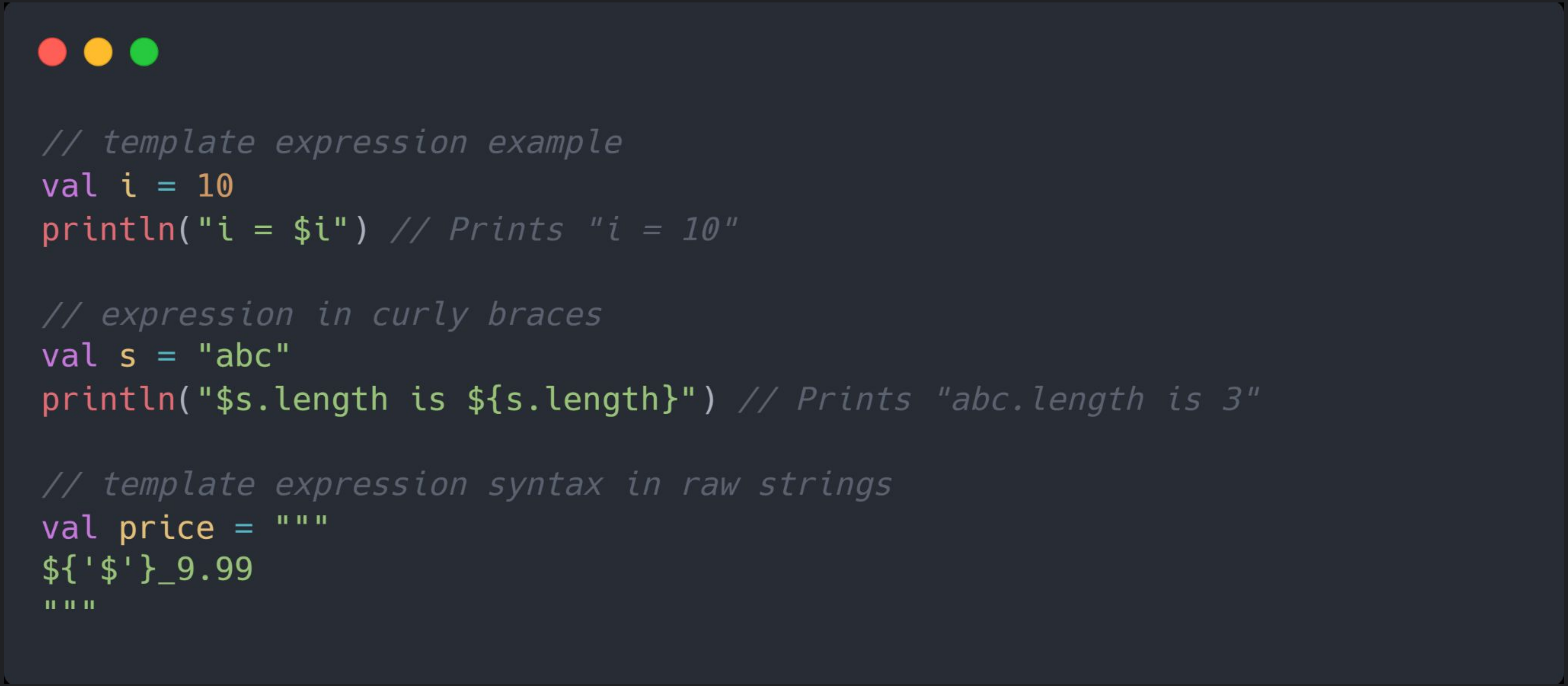
String Templates

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.



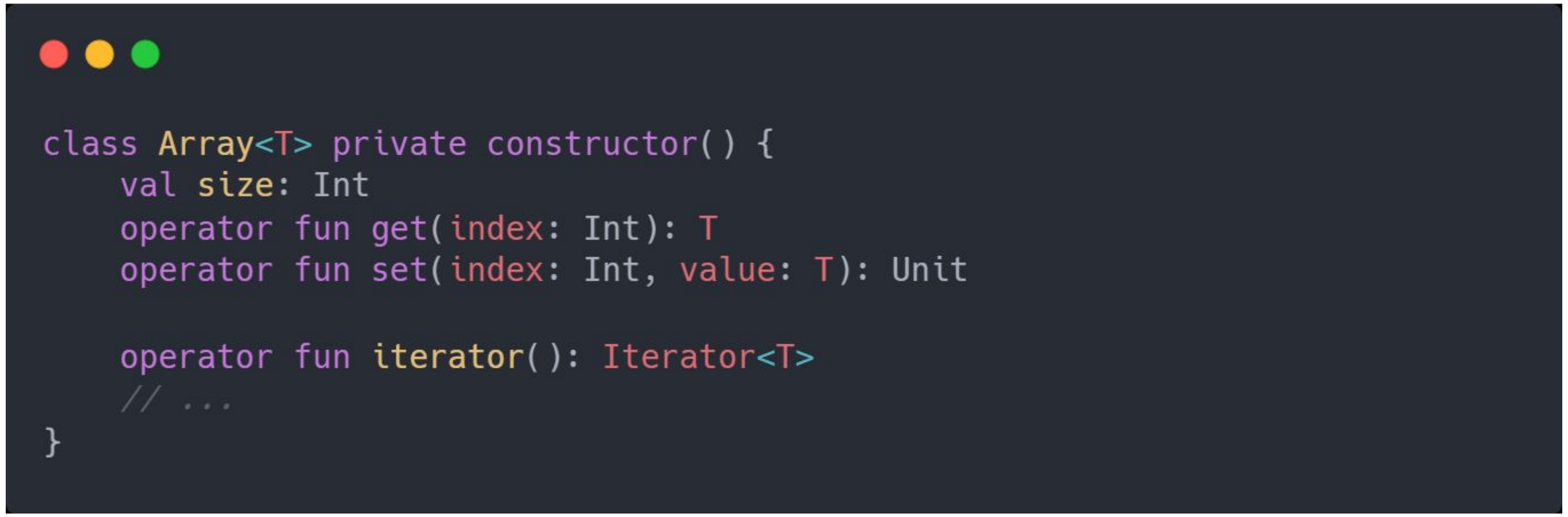
```kotlin
// template expression example
val i = 10
println("i = $i") // Prints "i = 10"

// 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:

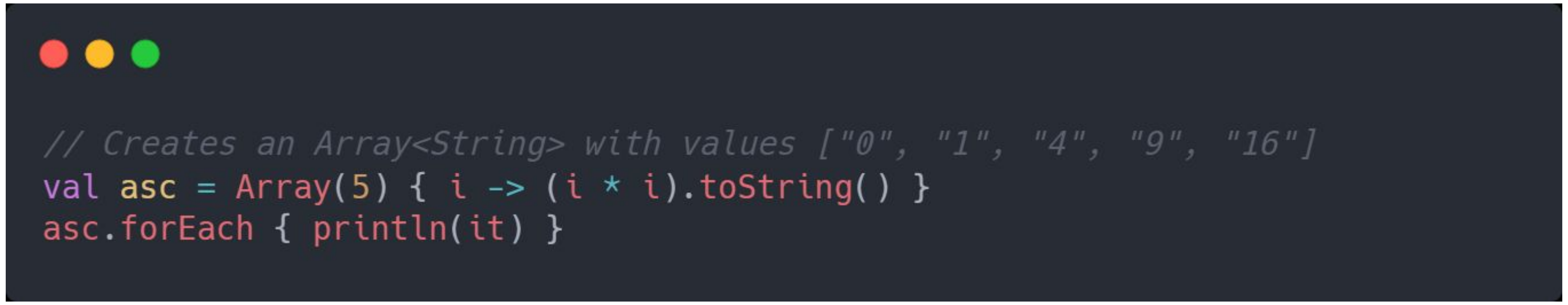


```
class Array<T> private constructor() {
    val size: Int
    operator fun get(index: Int): T
    operator fun set(index: Int, value: T): Unit

    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

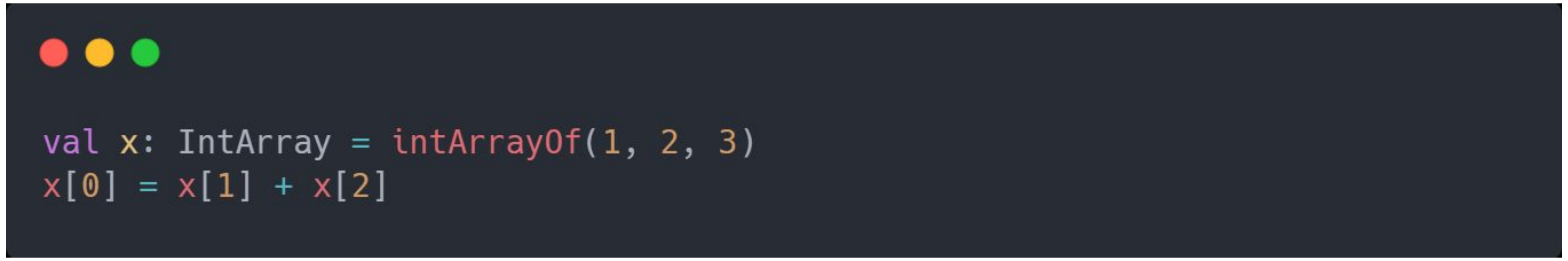
Arrays in Kotlin are ***invariant***.

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).

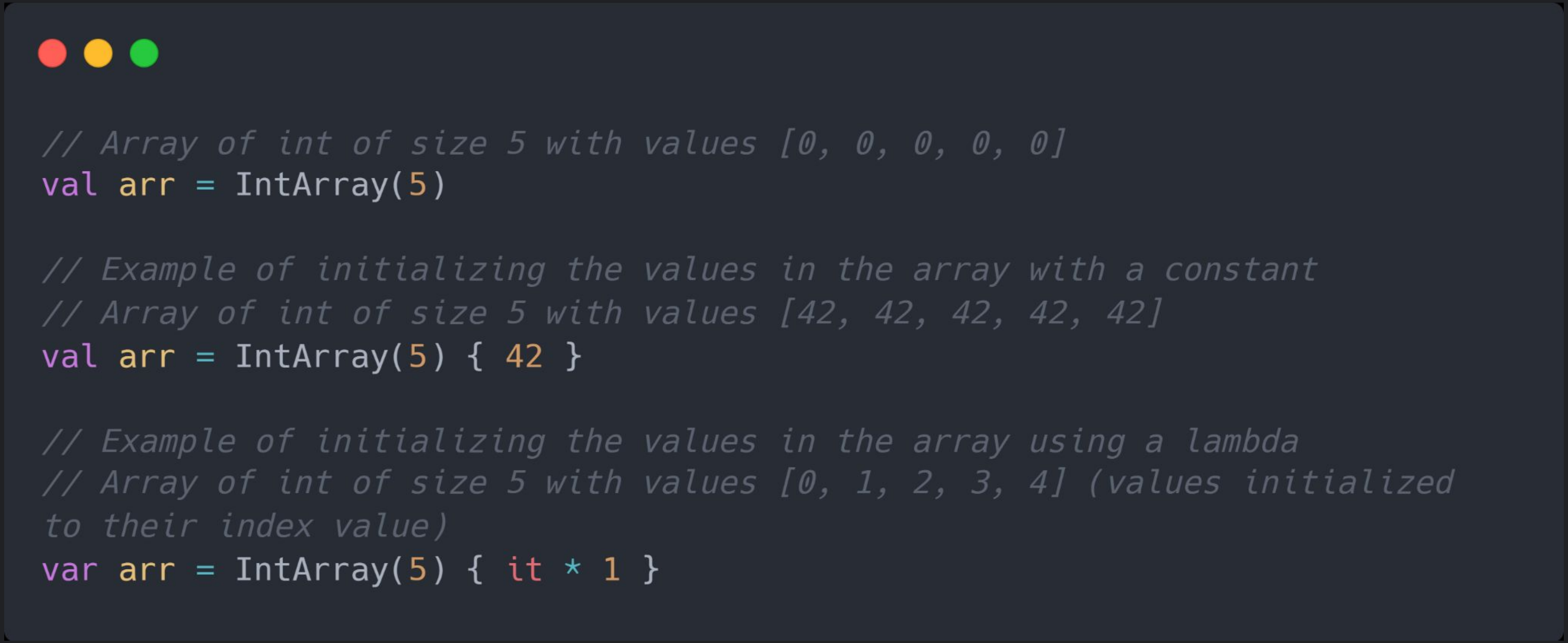
Primitive Type Arrays

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]
```



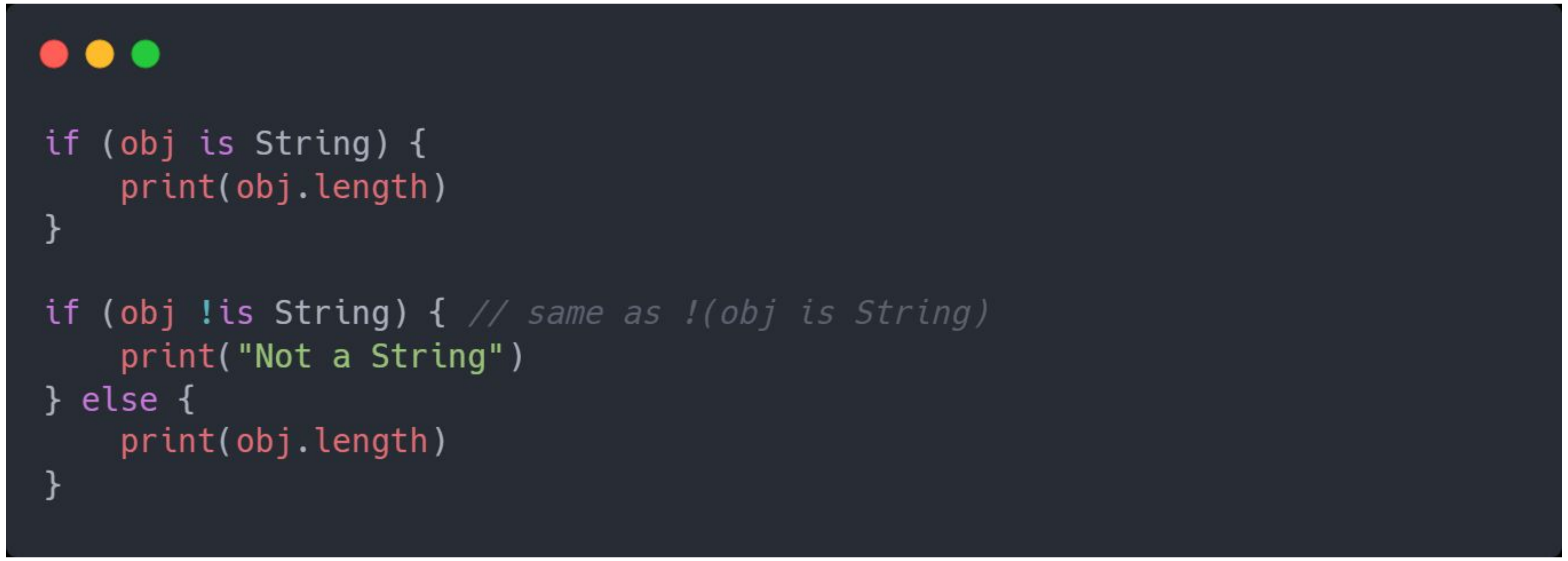
```
// Array of int of size 5 with values [0, 0, 0, 0, 0]
val arr = IntArray(5)

// 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

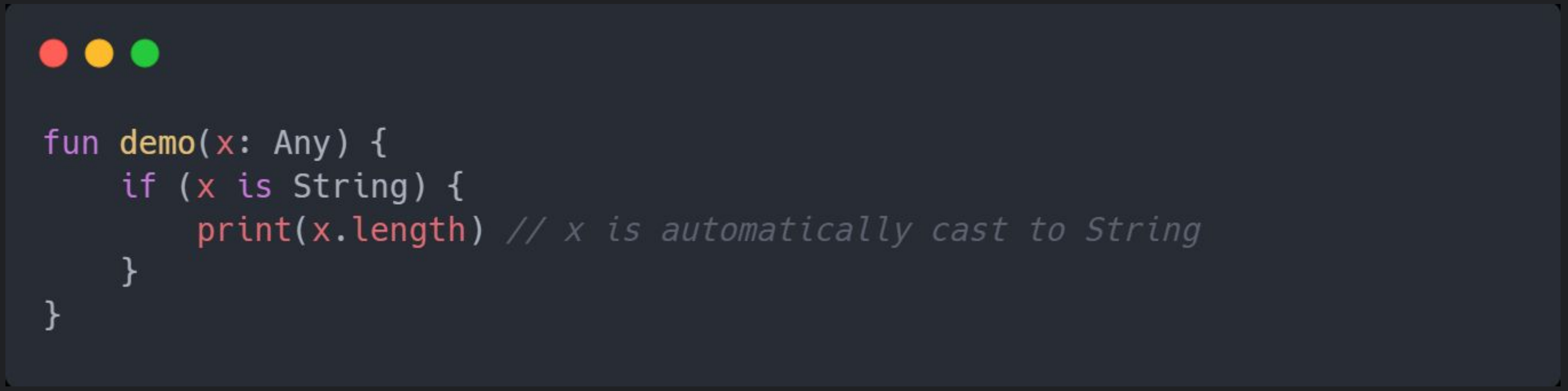
The is & !is operators

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



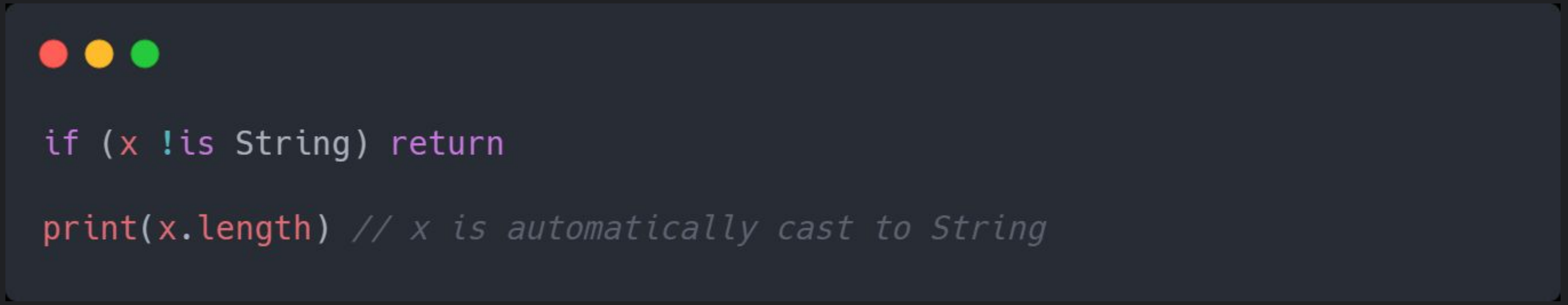
```
if (obj is String) {
    print(obj.length)
}

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:



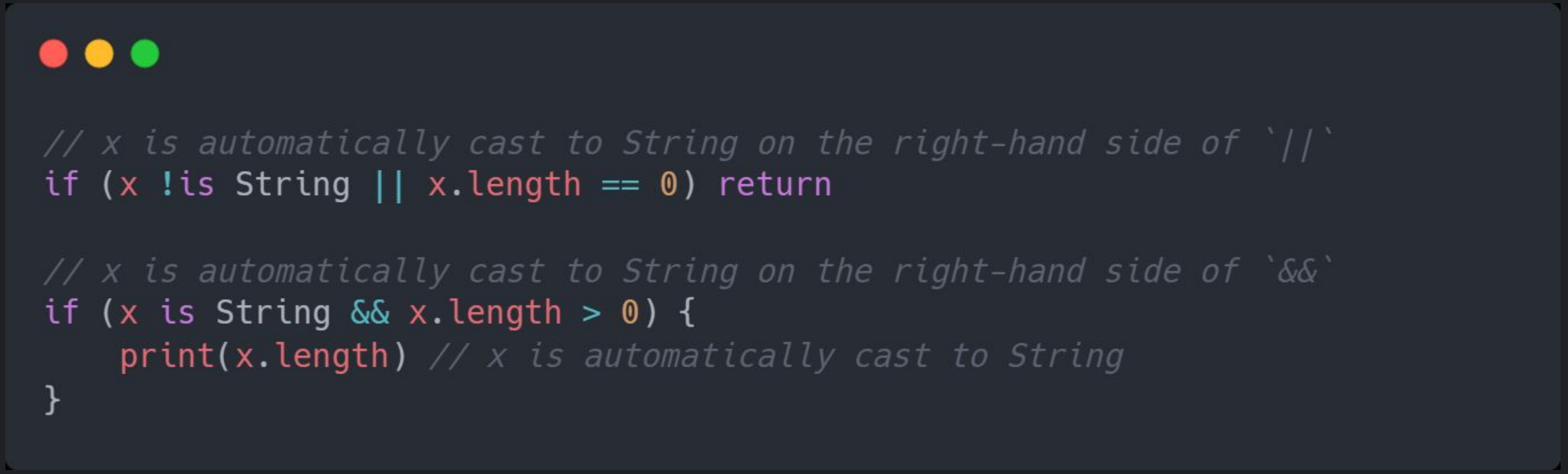
```kotlin
fun demo(x: Any) {
    if (x is String) {
        print(x.length) // x is automatically cast to String
    }
}
```

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



```kotlin
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:



```kotlin
// 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.

```kotlin
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:

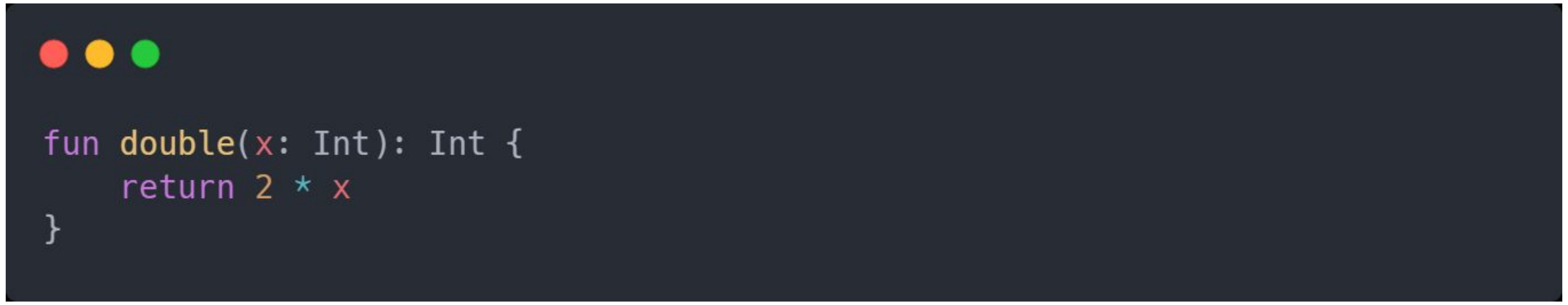
```kotlin
val x: String? = y as String?
```

Functions

Developer Student Clubs

Kotlin Functions

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



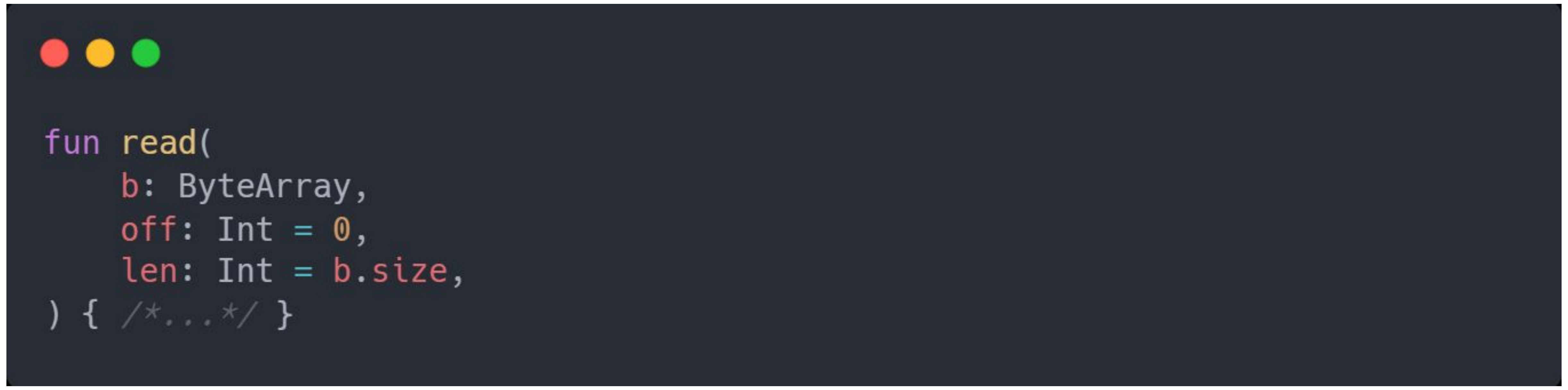
```
fun double(x: Int): Int {
    return 2 * x
}
```

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.

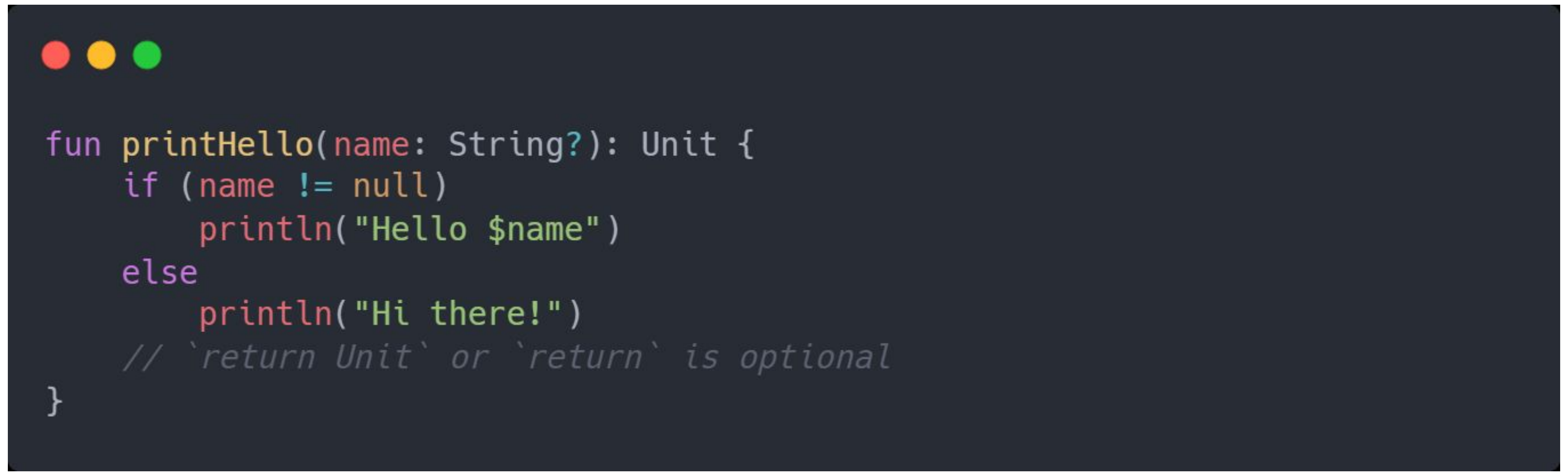
Function parameters can have default values, which are used when you skip the corresponding argument. This reduces the number of overloads:



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

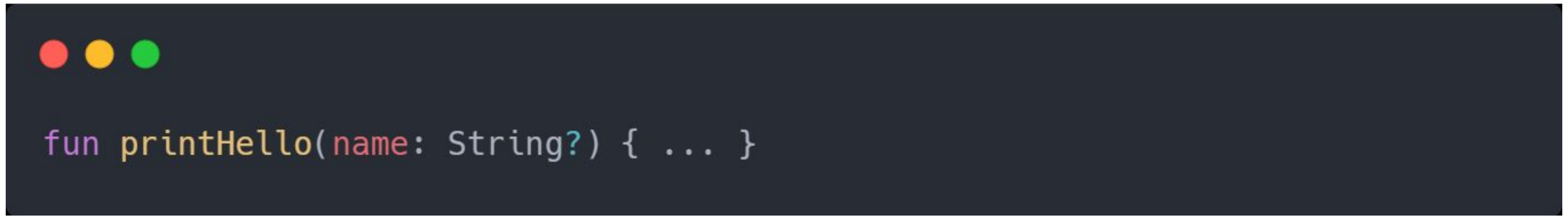
Unit Returning Functions

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:



```kotlin
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:



```kotlin
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
```

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!