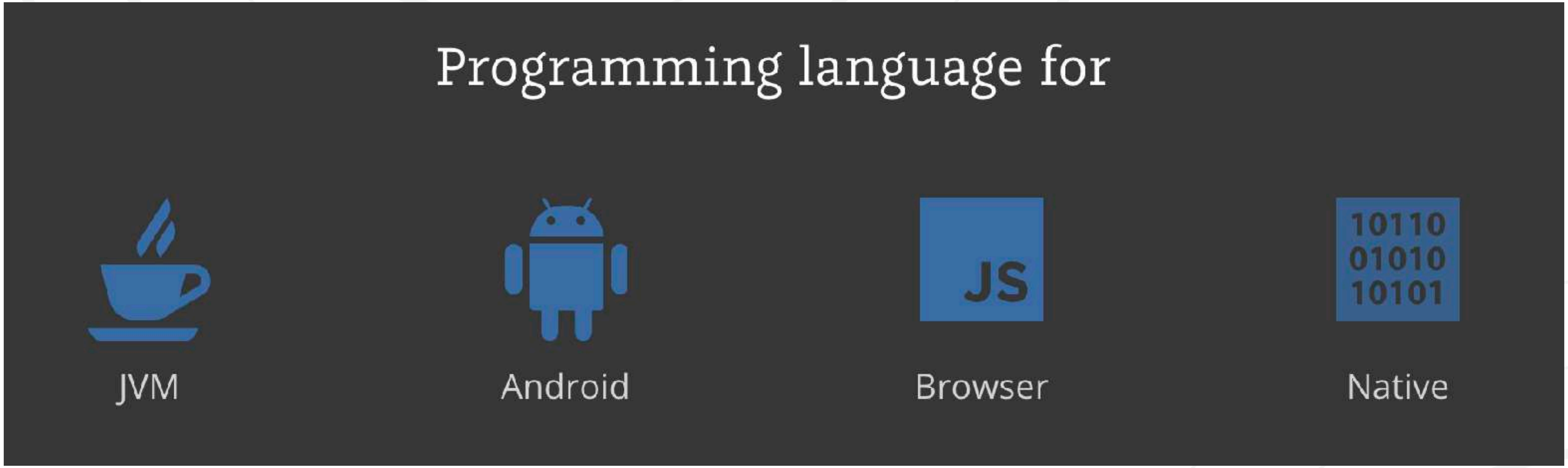
Introducción a Kotlin

emmanuel.lagarrigue@cs.uns.edu.ar

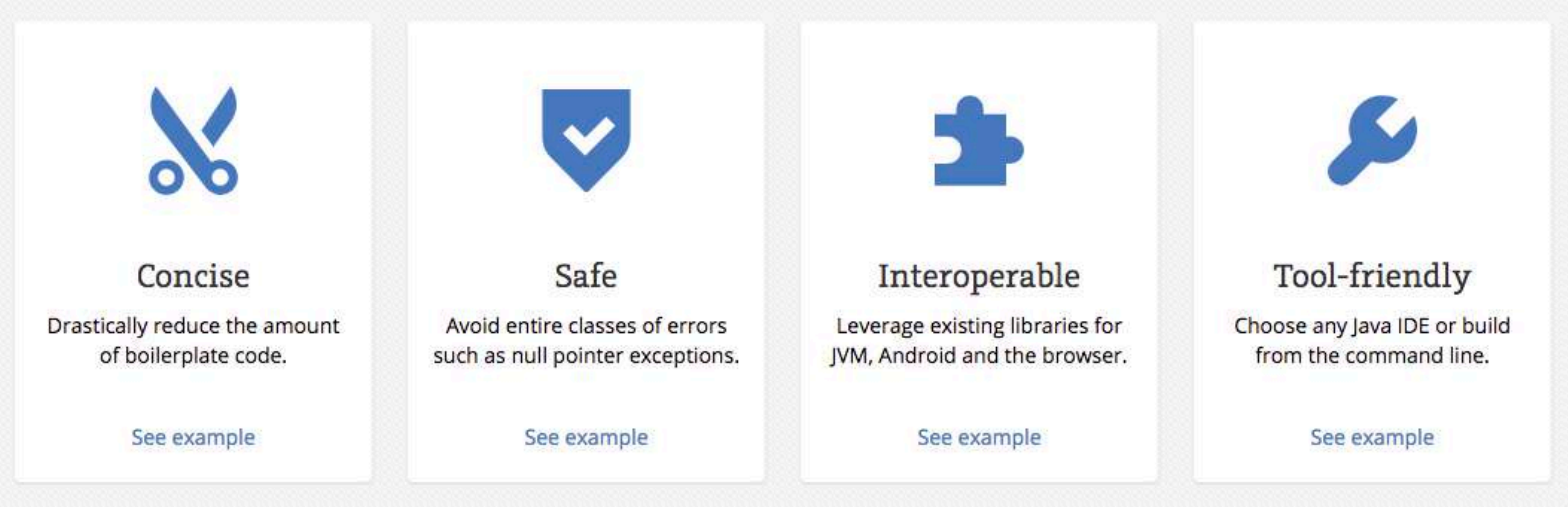
¿Qué es Kotlin?



Programming language for

https://kotlinlang.org/

¿Por qué Kotlin?



Concise

Drastically reduce the amount of boilerplate code.

See example

Safe

Avoid entire classes of errors such as null pointer exceptions.

See example

Interoperable

Leverage existing libraries for JVM, Android and the browser.

See example

Tool-friendly

Choose any Java IDE or build from the command line.

See example

¿Cómo es Kotlin?

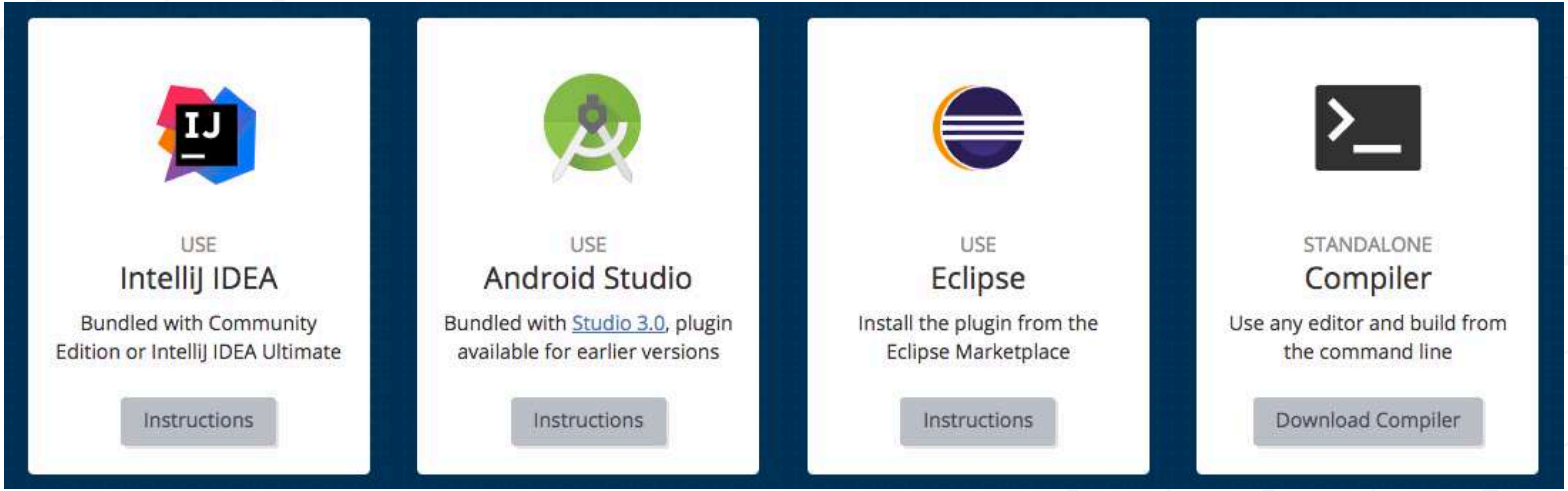
- Kotlin, al igual que Java, es estáticamente tipado.
- Kotlin, a diferencia de Java, tiene inferencia de tipos.

¿Cómo es Kotlin?

- Soporte de tipos Nullable
- Orientado a Objetos
- Soporte de Programación Funcional:
  - Function Types
  - Lambda expressions
  - Conjunto de APIs para trabajar con objetos y colecciones

¿Cómo es Kotlin?

- Gratis y Open Source http://github.com/ jetbrains/ kotlin
- https://try.kotl.in

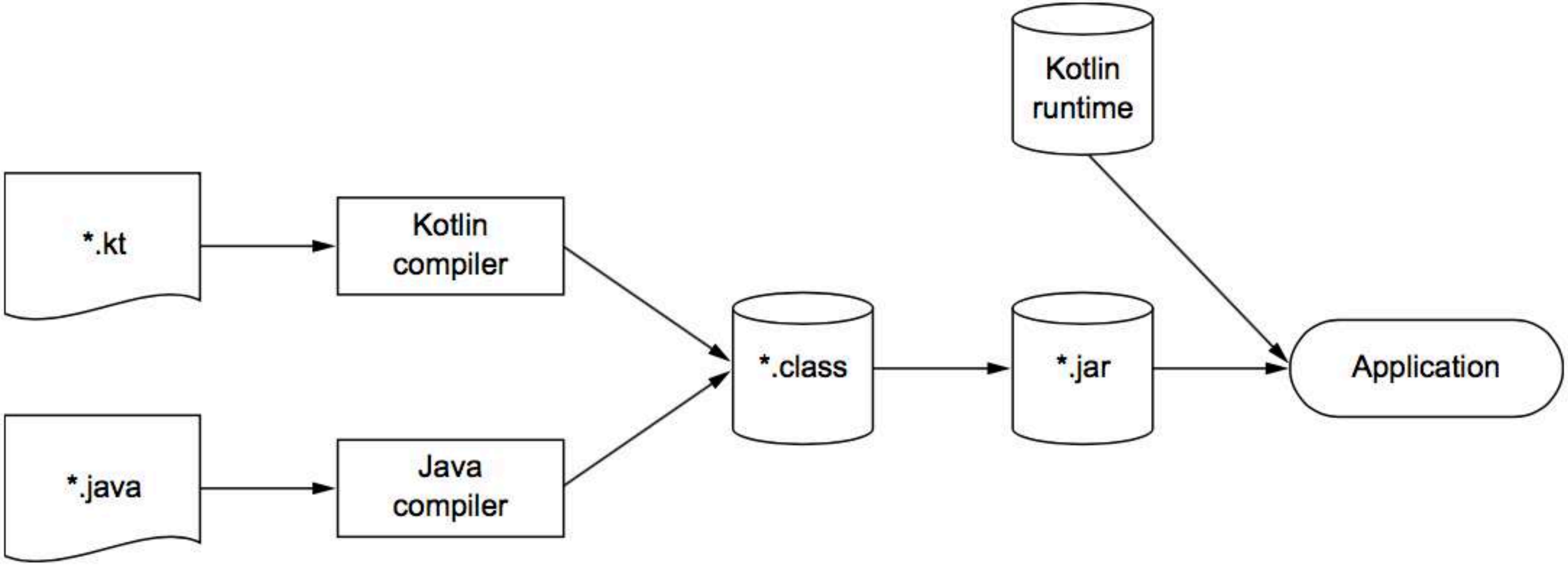


USE
IntelliJ IDEA
Bundled with Community Edition or IntelliJ IDEA Ultimate

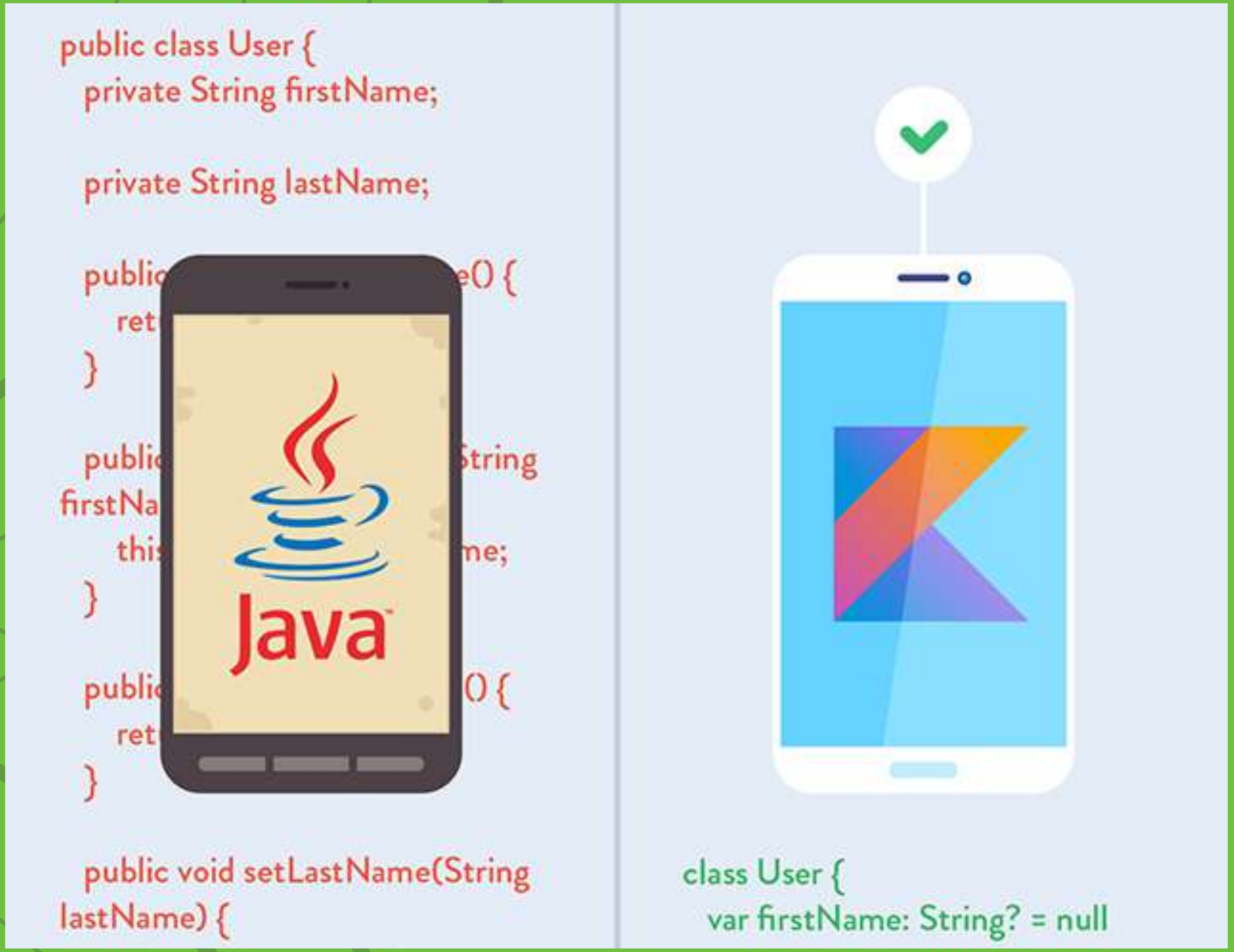
USE
Android Studio
Bundled with Studio 3.0, plugin available for earlier versions

USE
Eclipse
Install the plugin from the Eclipse Marketplace

STANDALONE
Compiler
Use any editor and build from the command line



Kotlin build process

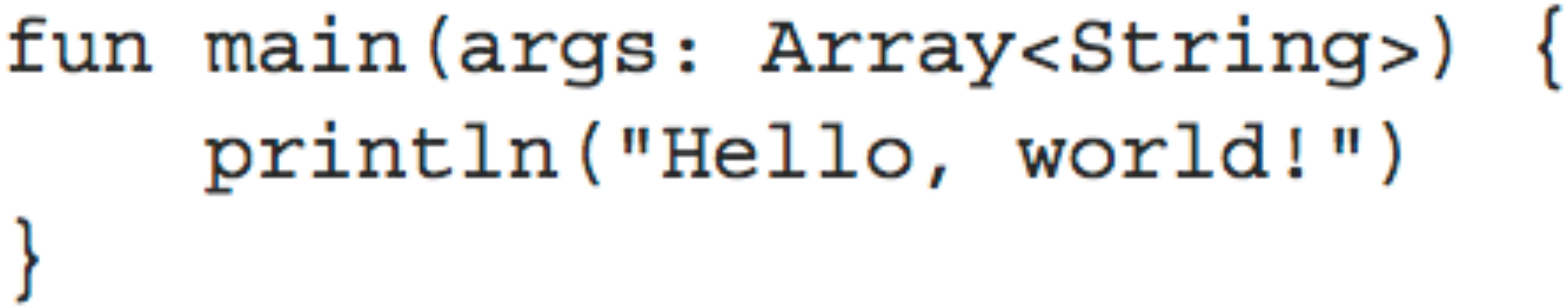


```
public class User {
  private String firstName;

  private String lastName;

  public                e() {
    ret
  }

  public                String
firstNa
    this                me;
  }

  public                () {
    ret
  }

  public void setLastName(String
lastName) {
```

```
class User {
  var firstName: String? = null
```

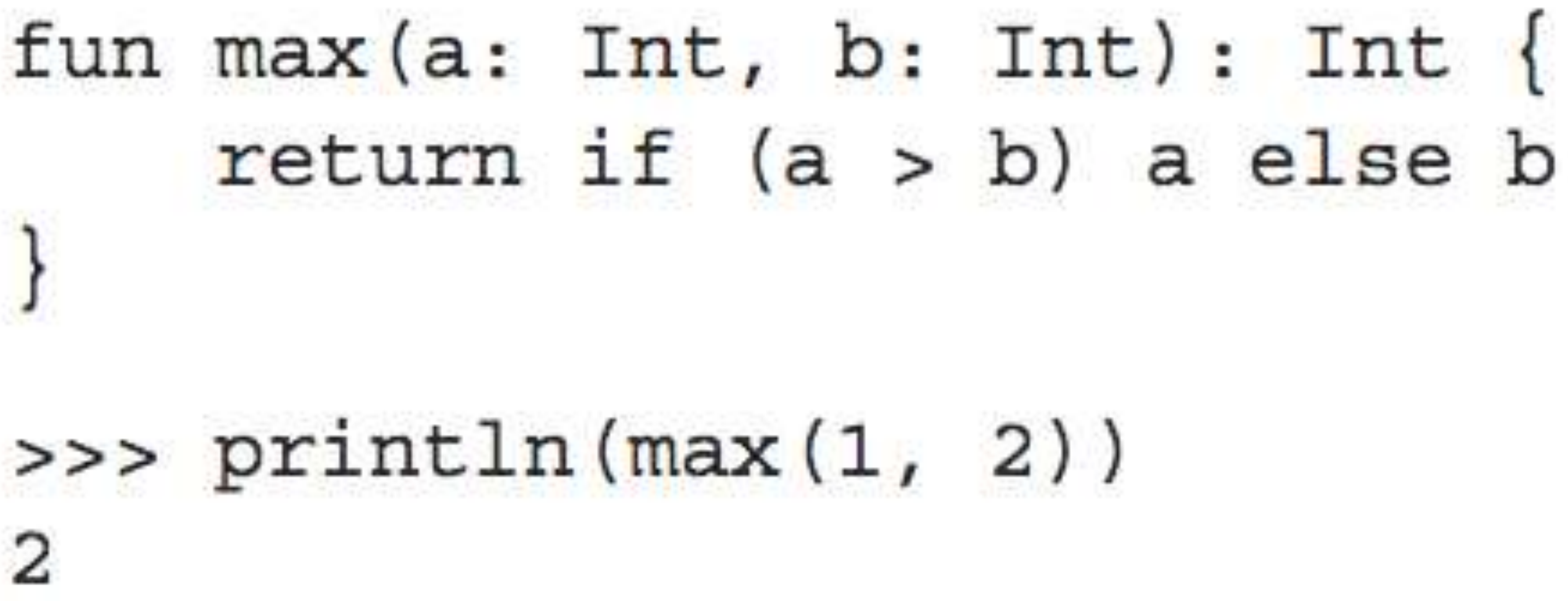
Kotlin Basics

Funciones y Variables

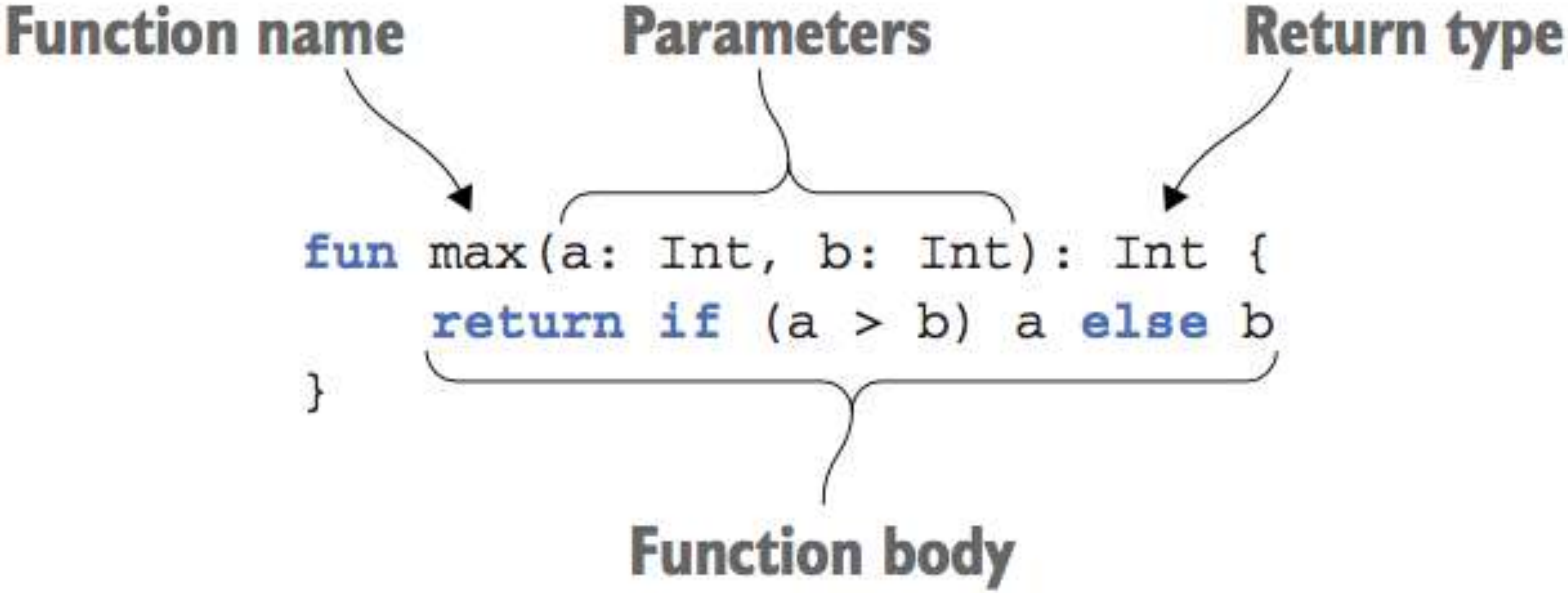


```
fun main(args: Array<String>) {
    println("Hello, world!")
}
```

Funciones



```
fun max(a: Int, b: Int): Int {
    return if (a > b) a else b
}

>>> println(max(1, 2))
2
```



Function name

Parameters

Return type

```
fun max(a: Int, b: Int): Int {
    return if (a > b) a else b
}
```

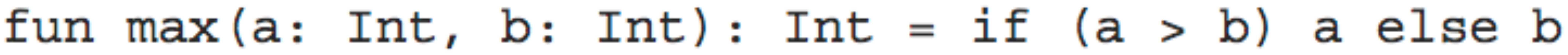
Function body

Sentencias y Expresiones

- En Kotlin, **if** es una expresión, no una sentencia
- Las expresiones tienen valor y pueden utilizarse en otras expresiones
- En Kotlin, la mayoría de las estructuras son expresiones, excepto por los loops

Cuerpos expresiones

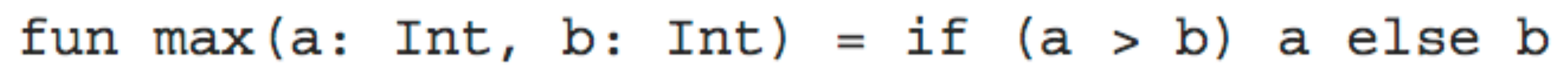
- Si el cuerpo de una función consiste de una sola expresión, se puede utilizar dicha expresión como cuerpo sin los corchetes



```
fun max(a: Int, b: Int): Int = if (a > b) a else b
```

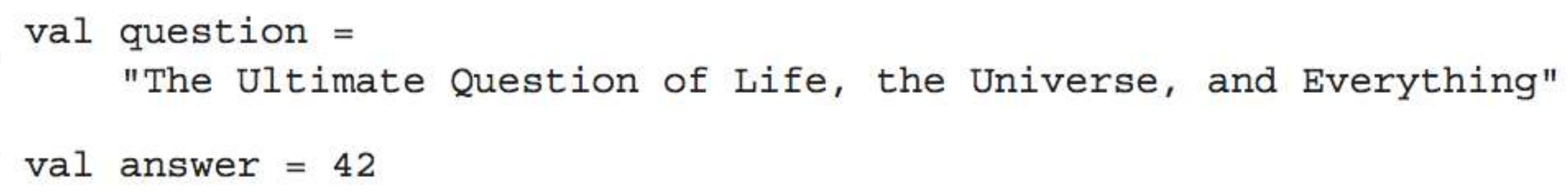
Cuerpos expresiones

- Se puede omitir el tipo, ya que es inferido por el tipo de la expresión

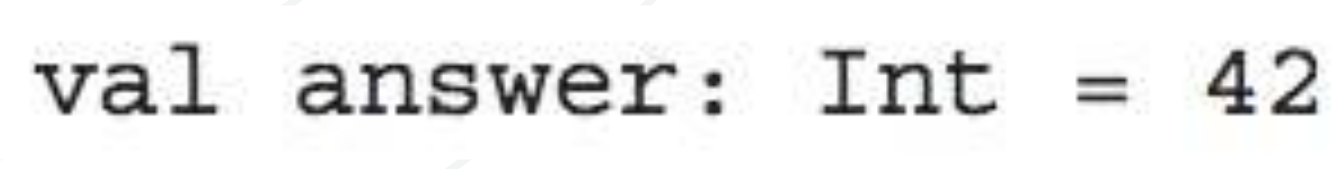


```
fun max(a: Int, b: Int) = if (a > b) a else b
```

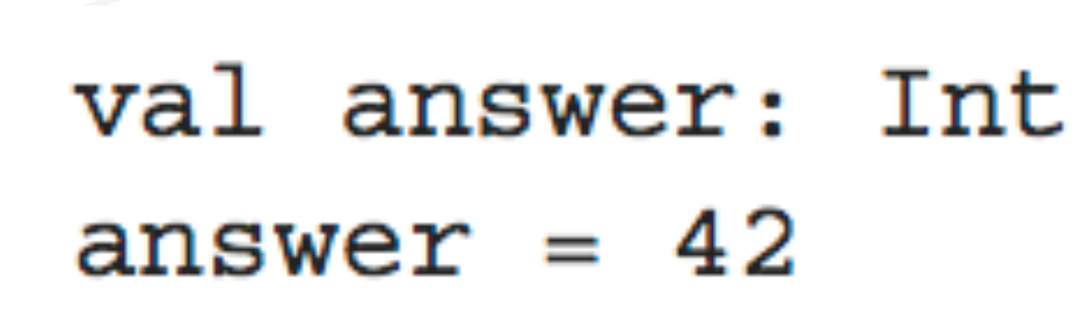
Variables



```
val question =
    "The Ultimate Question of Life, the Universe, and Everything"

val answer = 42
```



```
val answer: Int = 42
```

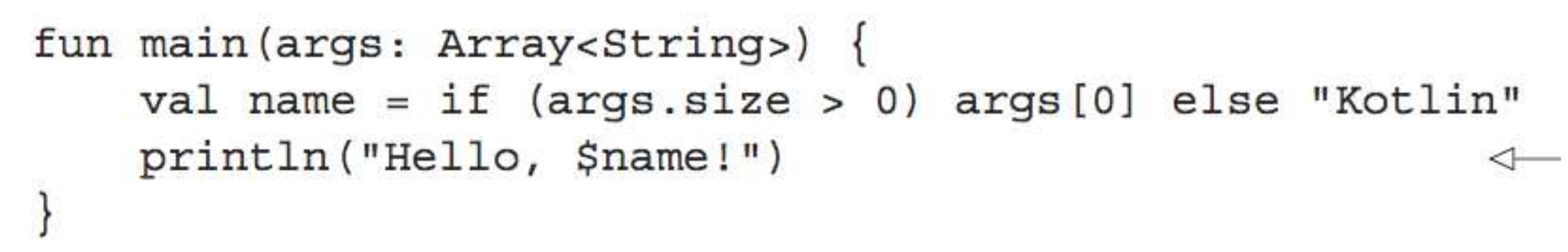


```
val answer: Int
answer = 42
```

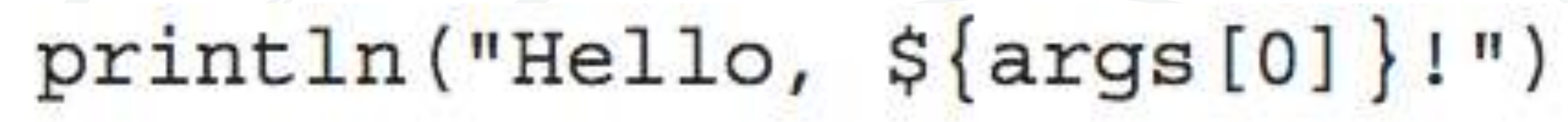
Variables Mutables e Inmutables

- **val (value)**: Referencia inmutable. Una variable declarada con **val** no puede ser re asignada luego de ser inicializada. Idem **final** en java.
- **var (variable)**: Referencia mutable. El valor de la variable puede cambiar. Idem declaración normal en java. El tipo no puede cambiar luego de la 1ra asignación.

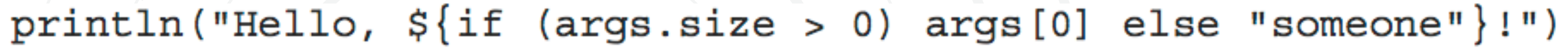
String templates



```
fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "Kotlin"
    println("Hello, $name!")
}
```

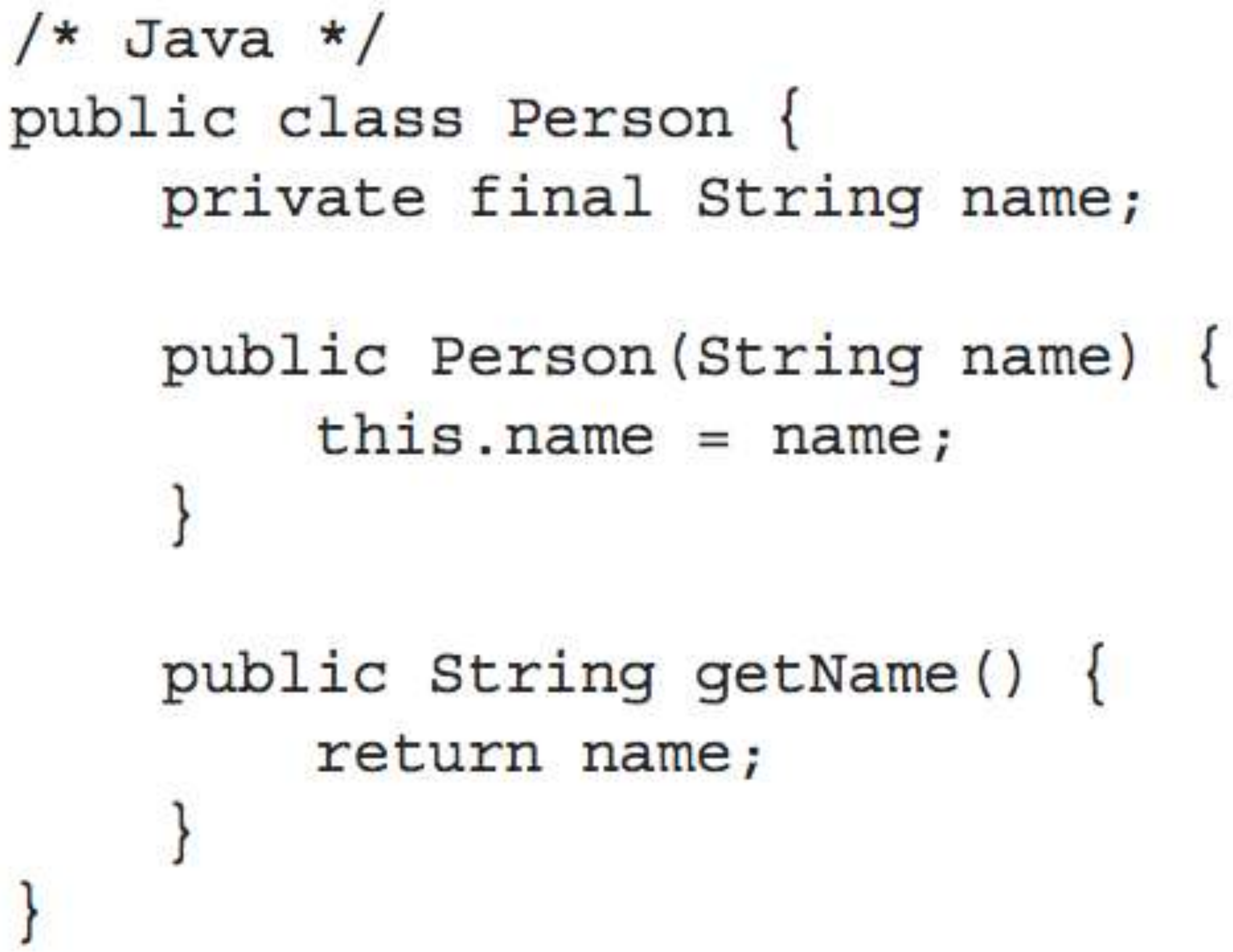


```
println("Hello, ${args[0]}!")
```



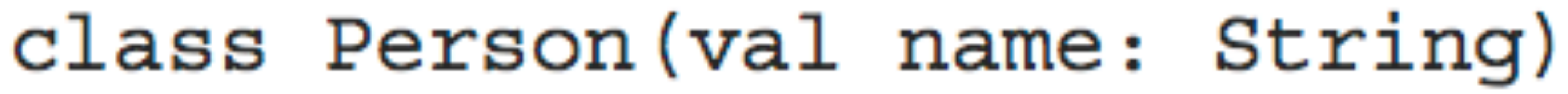
```
println("Hello, ${if (args.size > 0) args[0] else "someone"}!")
```

Clases y propiedades



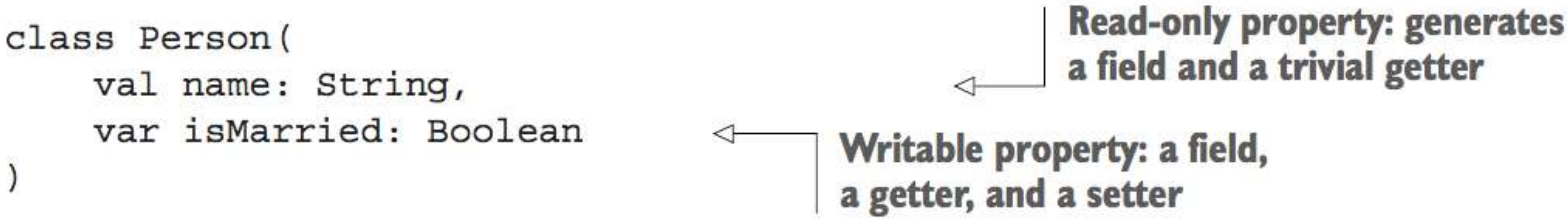
```
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

Kotlin



```
class Person(val name: String)
```

Propiedades



```
class Person(
    val name: String,
    var isMarried: Boolean
)
```

Read-only property: generates a field and a trivial getter

Writable property: a field, a getter, and a setter

```
/* Java */
>>> Person person = new Person("Bob", true);
>>> System.out.println(person.getName());
Bob
>>> System.out.println(person.isMarried());
true
```

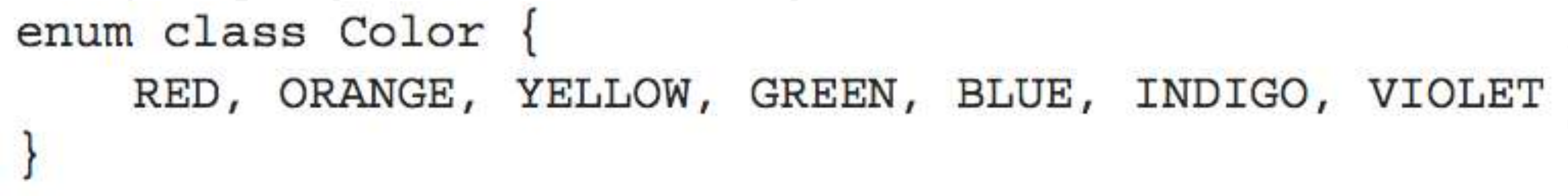
Kotlin

```
>>> val person = Person("Bob", true)
>>> println(person.name)
Bob
>>> println(person.isMarried)
true
```

Call the constructor without the "new" keyword.

You access the property directly, but the getter is invoked.

Enums



```
enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

Enums

```
enum class Color(
        val r: Int, val g: Int, val b: Int
) {
    RED(255, 0, 0), ORANGE(255, 165, 0),
    YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),
    INDIGO(75, 0, 130), VIOLET(238, 130, 238);

    fun rgb() = (r * 256 + g) * 256 + b
}
>>> println(Color.BLUE.rgb())
255
```

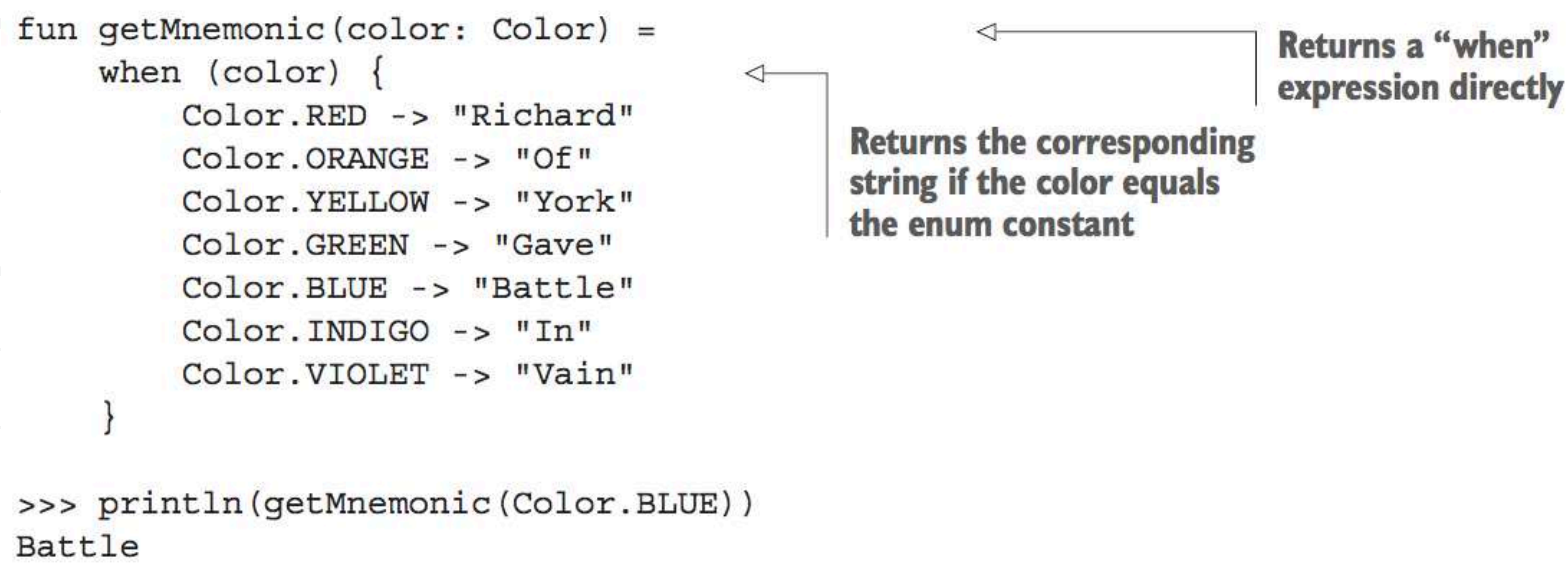
Declares properties of enum constants

Specifies property values when each constant is created

The semicolon here is required.

Defines a method on the enum class

When enums



```
fun getMnemonic(color: Color) =
    when (color) {
        Color.RED -> "Richard"
        Color.ORANGE -> "Of"
        Color.YELLOW -> "York"
        Color.GREEN -> "Gave"
        Color.BLUE -> "Battle"
        Color.INDIGO -> "In"
        Color.VIOLET -> "Vain"
    }

>>> println(getMnemonic(Color.BLUE))
Battle
```

Returns a "when" expression directly

Returns the corresponding string if the color equals the enum constant

When cualquier objeto

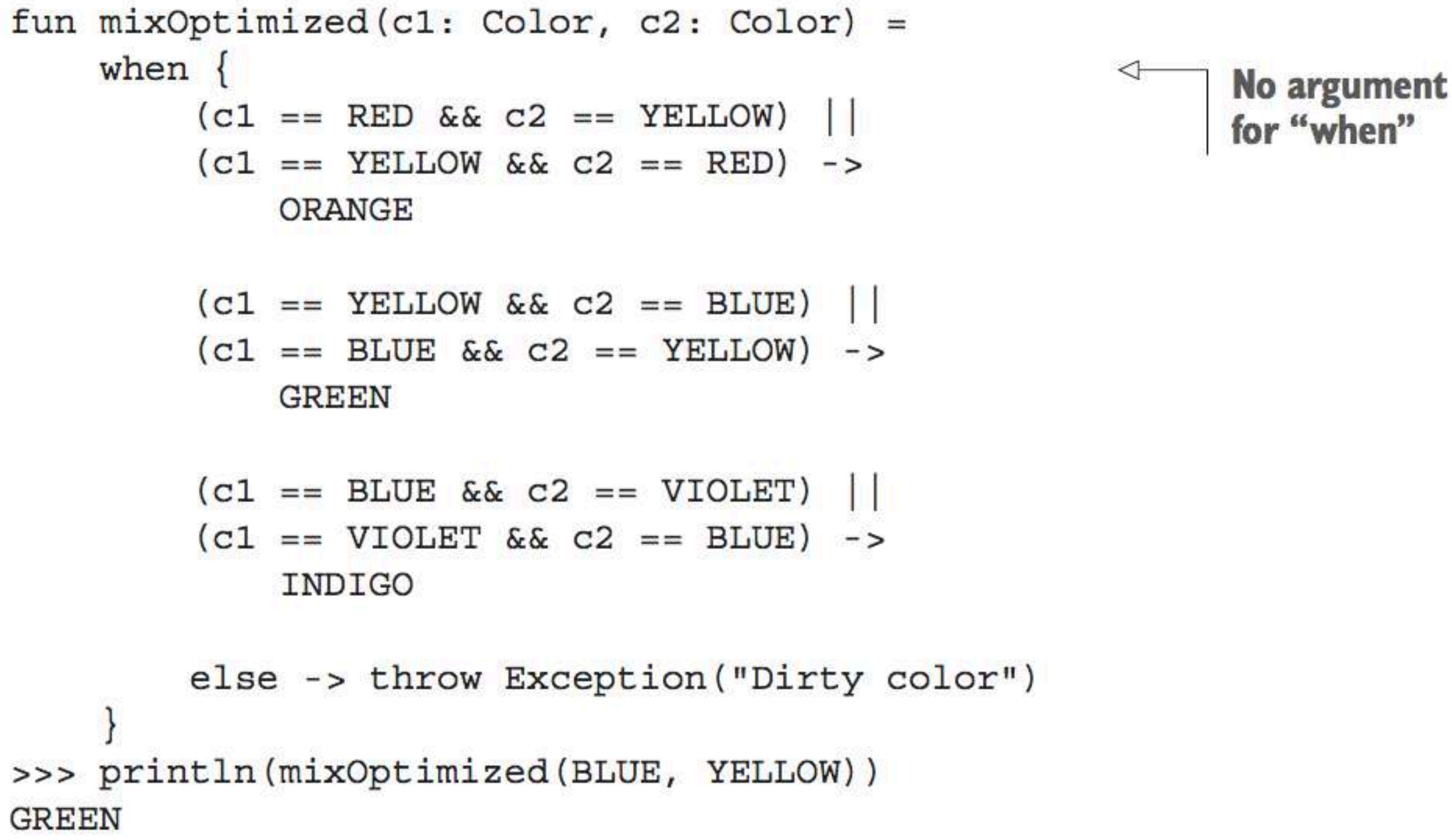
```
fun mix(c1: Color, c2: Color) =
        when (setOf(c1, c2)) {
            setOf(RED, YELLOW) -> ORANGE
            setOf(YELLOW, BLUE) -> GREEN
            setOf(BLUE, VIOLET) -> INDIGO
            else -> throw Exception("Dirty color")
        }

>>> println(mix(BLUE, YELLOW))
GREEN
```

An argument of the "when" expression can be any object. It's checked for equality with the branch conditions.

Enumerates pairs of colors that can be mixed

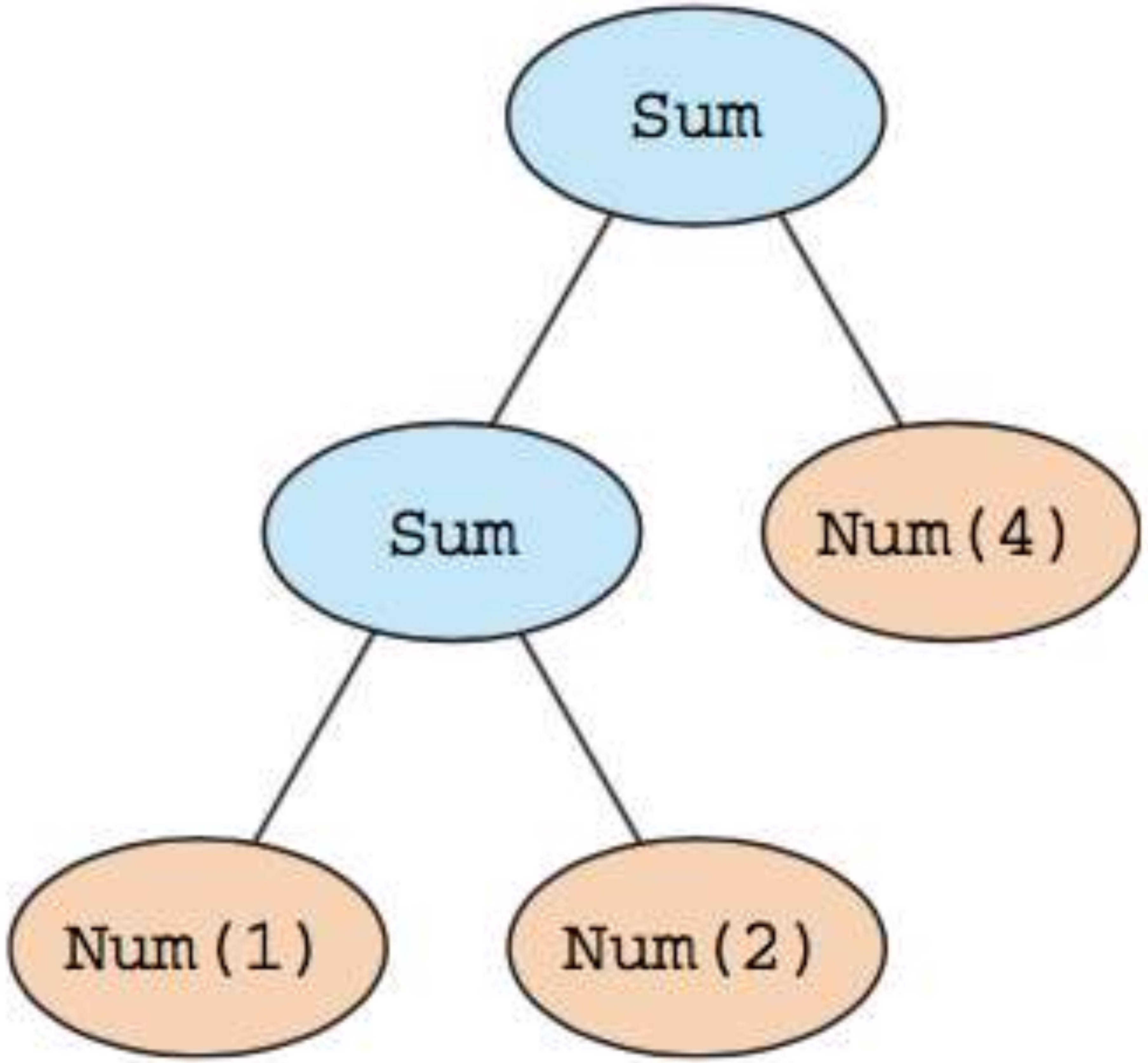
Executed if none of the other branches were matched

When nada

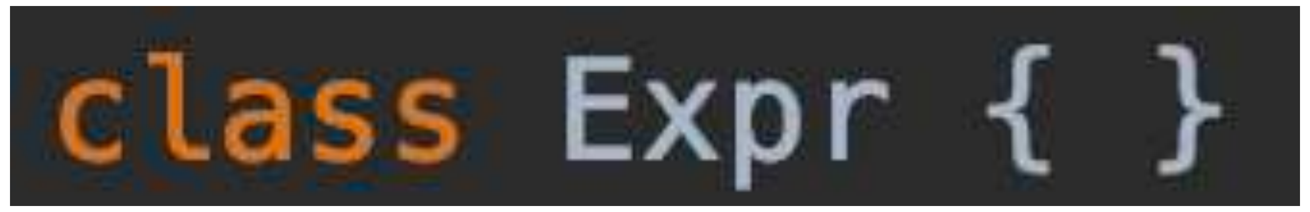


```
fun mixOptimized(c1: Color, c2: Color) =
    when {
        (c1 == RED && c2 == YELLOW) ||
        (c1 == YELLOW && c2 == RED) ->
            ORANGE

        (c1 == YELLOW && c2 == BLUE) ||
        (c1 == BLUE && c2 == YELLOW) ->
            GREEN

        (c1 == BLUE && c2 == VIOLET) ||
        (c1 == VIOLET && c2 == BLUE) ->
            INDIGO

        else -> throw Exception("Dirty color")
    }
>>> println(mixOptimized(BLUE, YELLOW))
GREEN
```

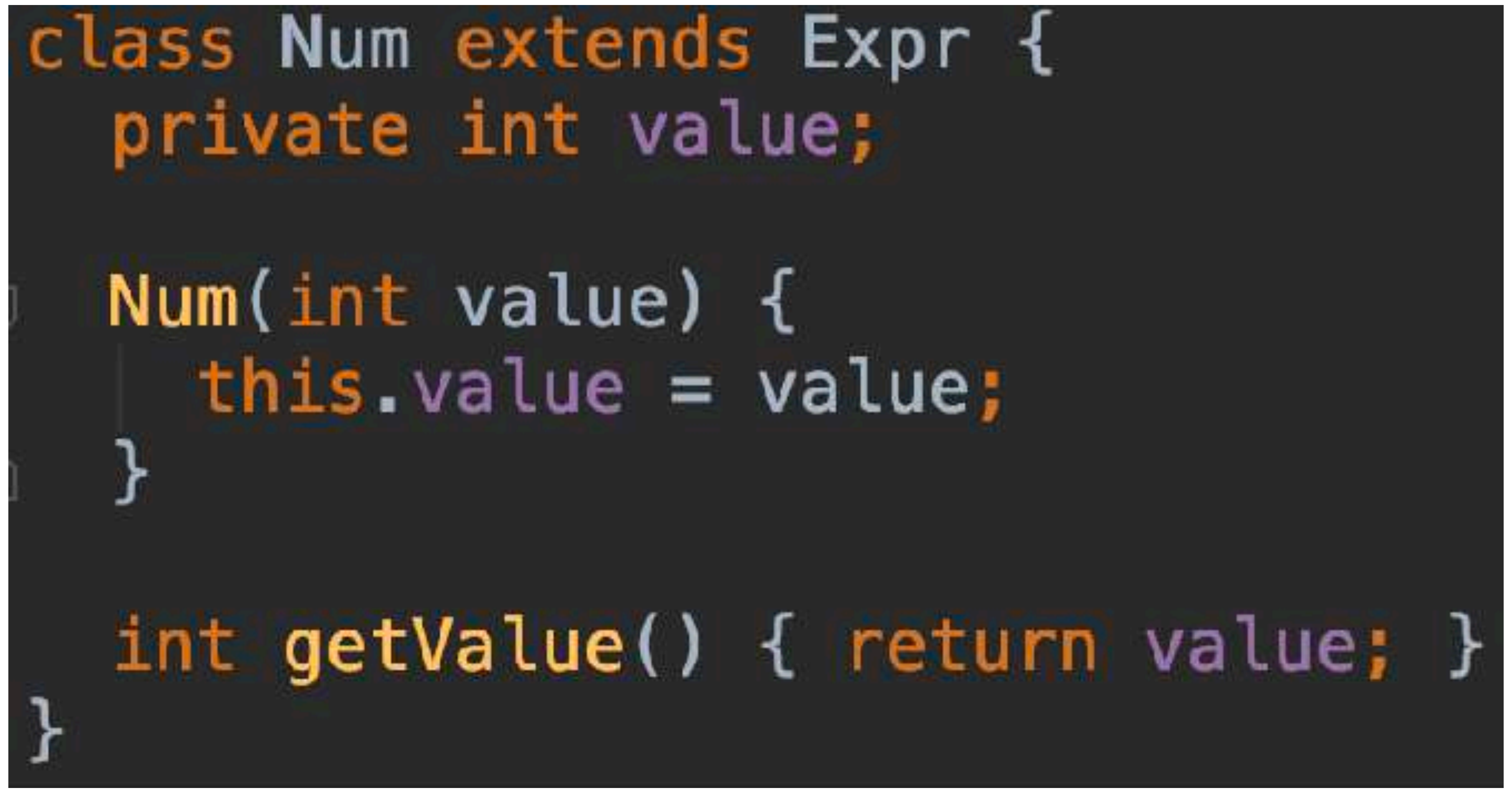
No argument for “when”



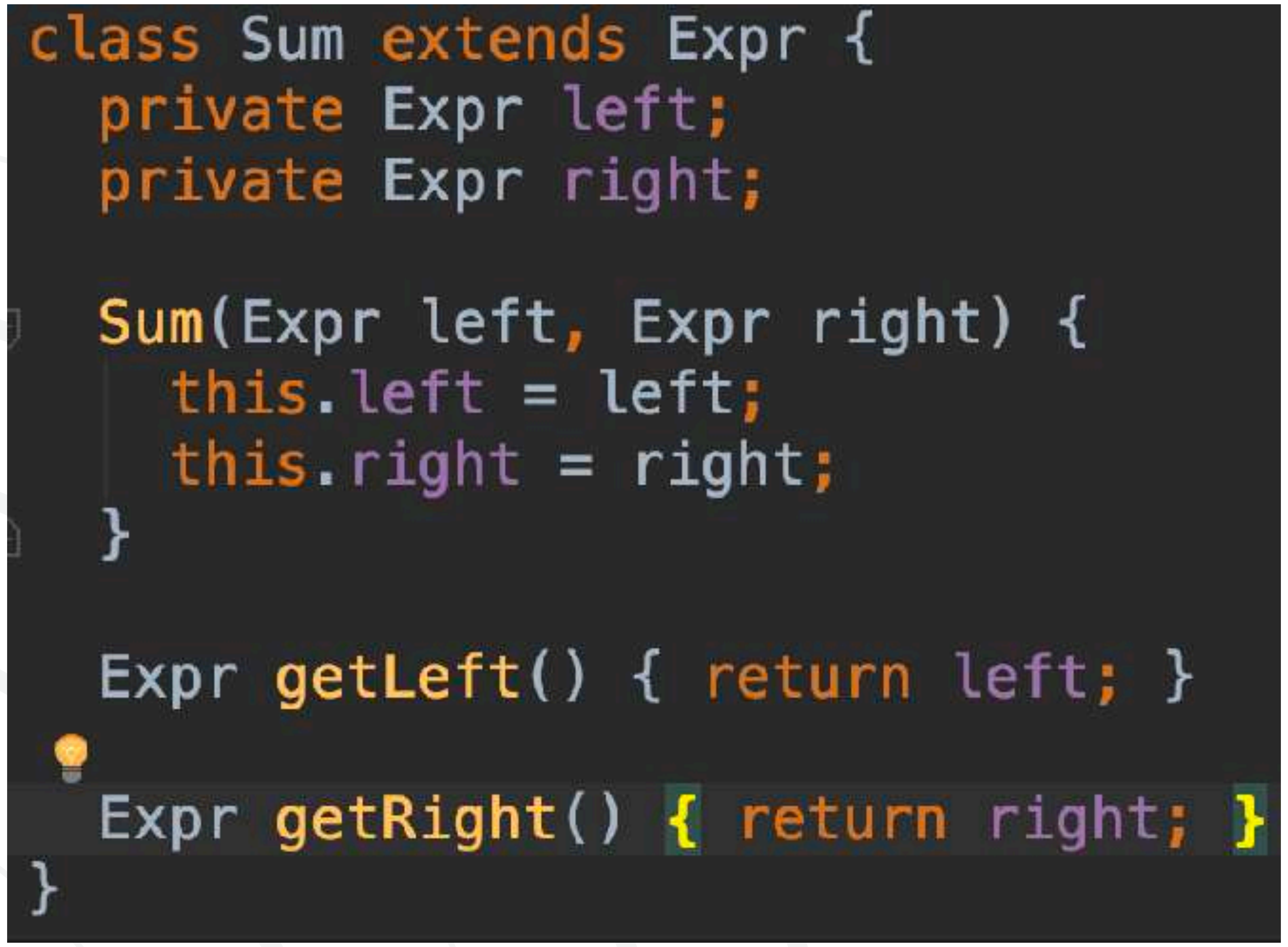
$(1 + 2) + 4$



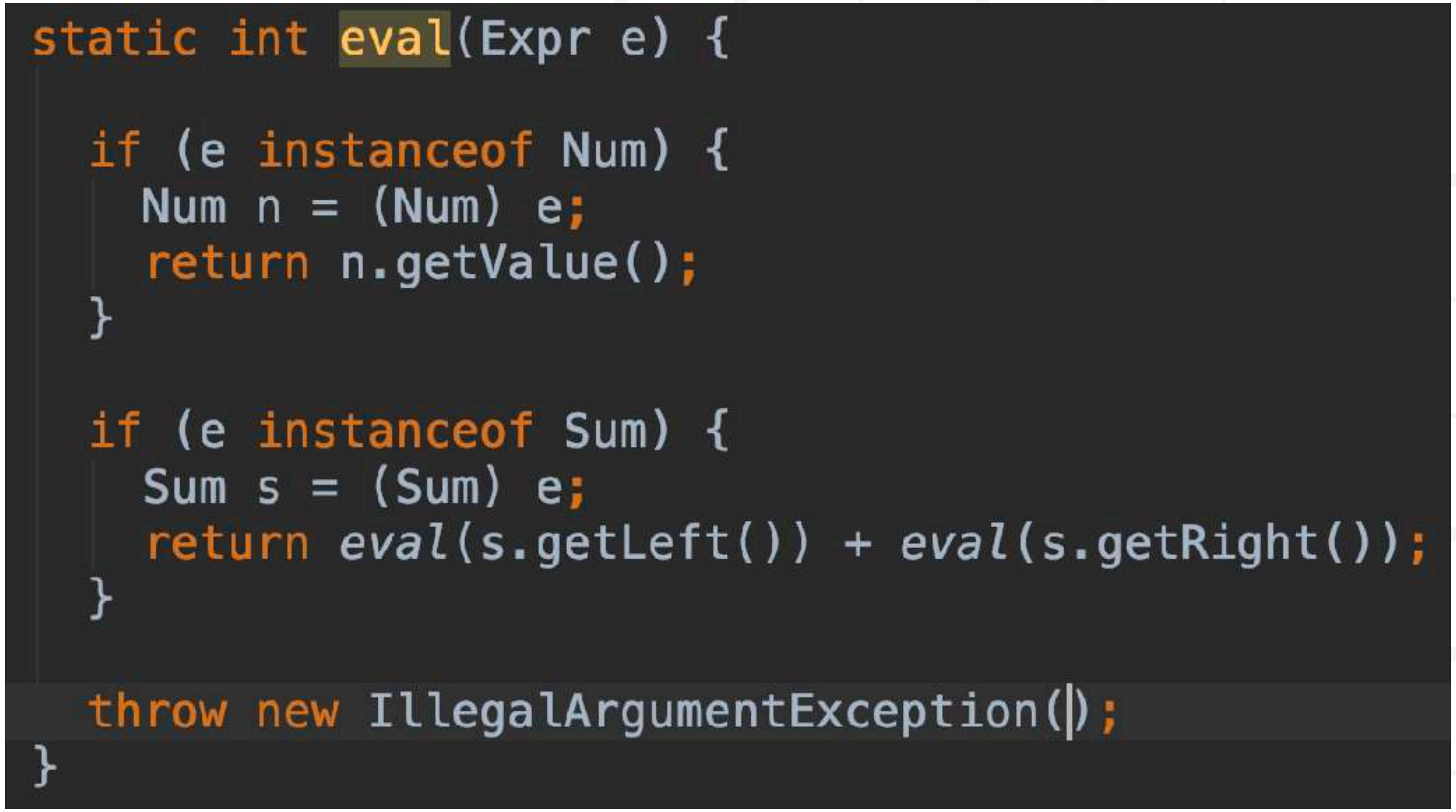
```
class Expr { }
```



```
class Num extends Expr {
  private int value;

  Num(int value) {
    this.value = value;
  }

  int getValue() { return value; }
}
```



```
class Sum extends Expr {
  private Expr left;
  private Expr right;

  Sum(Expr left, Expr right) {
    this.left = left;
    this.right = right;
  }

  Expr getLeft() { return left; }

  Expr getRight() { return right; }
}
```



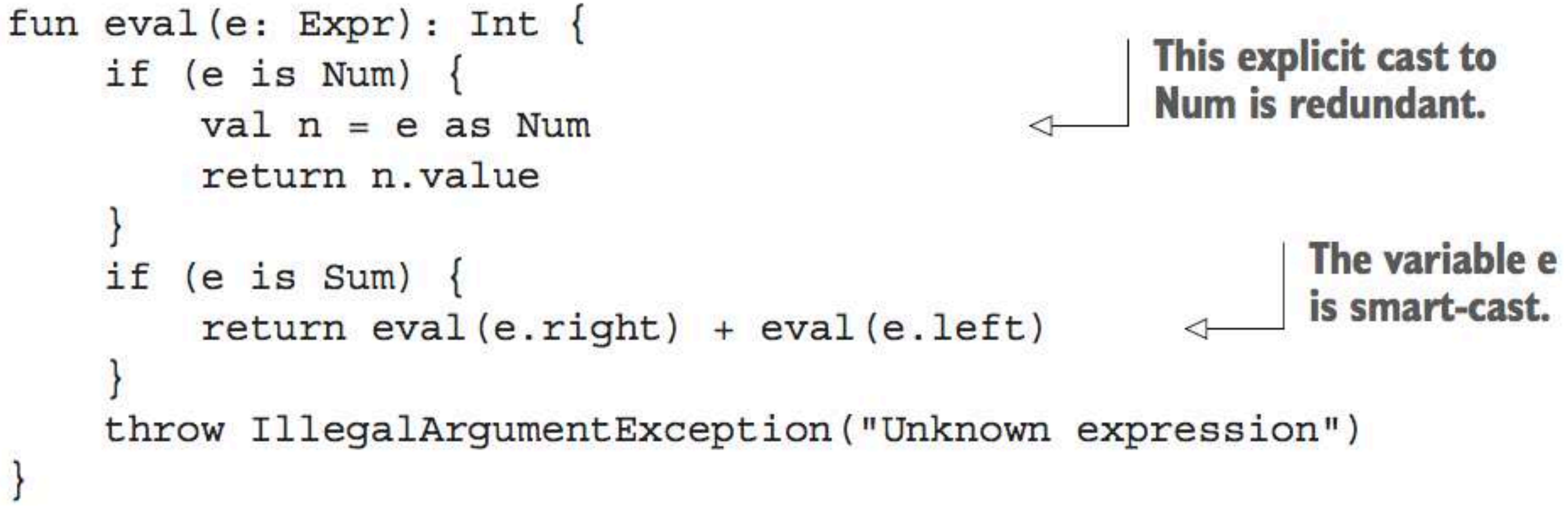
```
static int eval(Expr e) {

  if (e instanceof Num) {
    Num n = (Num) e;
    return n.getValue();
  }

  if (e instanceof Sum) {
    Sum s = (Sum) e;
    return eval(s.getLeft()) + eval(s.getRight());
  }

  throw new IllegalArgumentException();
}
```

Smart casts: combinando checkeo de tipos y cast

```
interface Expr
class Num(val value: Int) : Expr
class Sum(val left: Expr, val right: Expr) : Expr
```

Simple value object class with one property, value, implementing the Expr interface

The argument of a Sum operation can be any Expr: either Num or another Sum

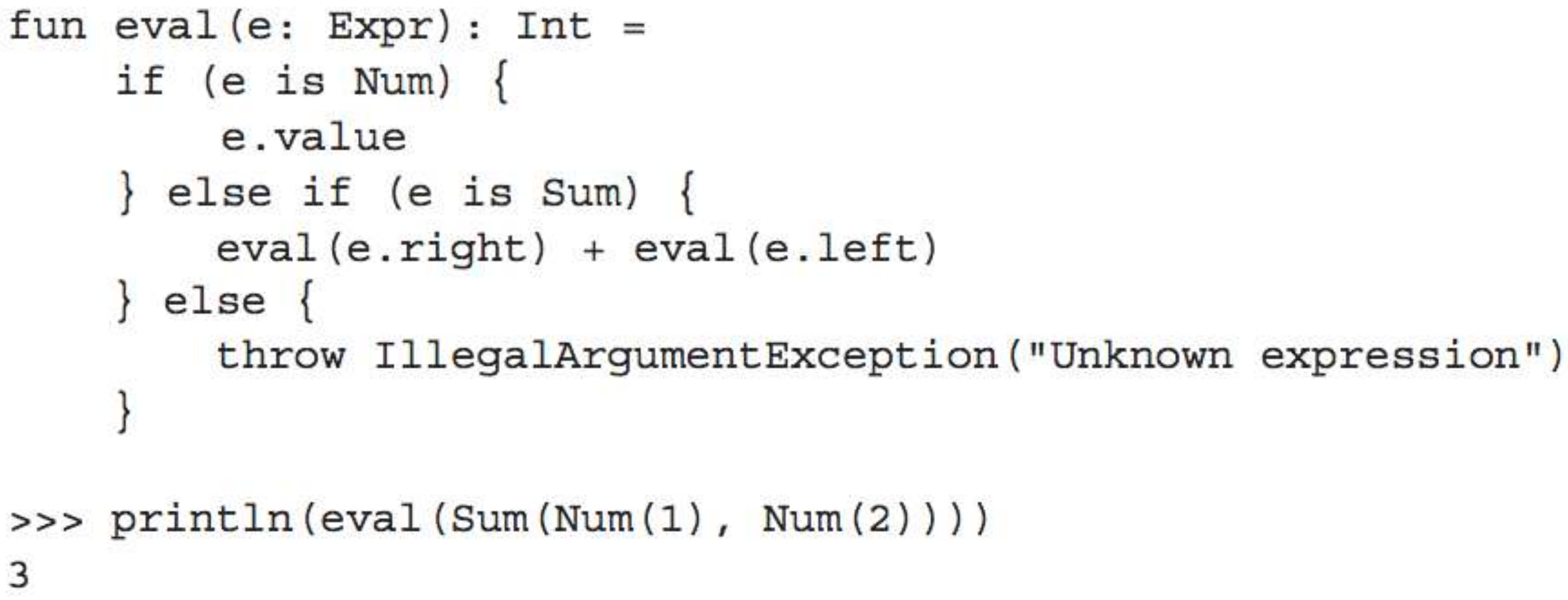


```
fun eval(e: Expr): Int {
    if (e is Num) {
        val n = e as Num
        return n.value
    }
    if (e is Sum) {
        return eval(e.right) + eval(e.left)
    }
    throw IllegalArgumentException("Unknown expression")
}
```

This explicit cast to Num is redundant.

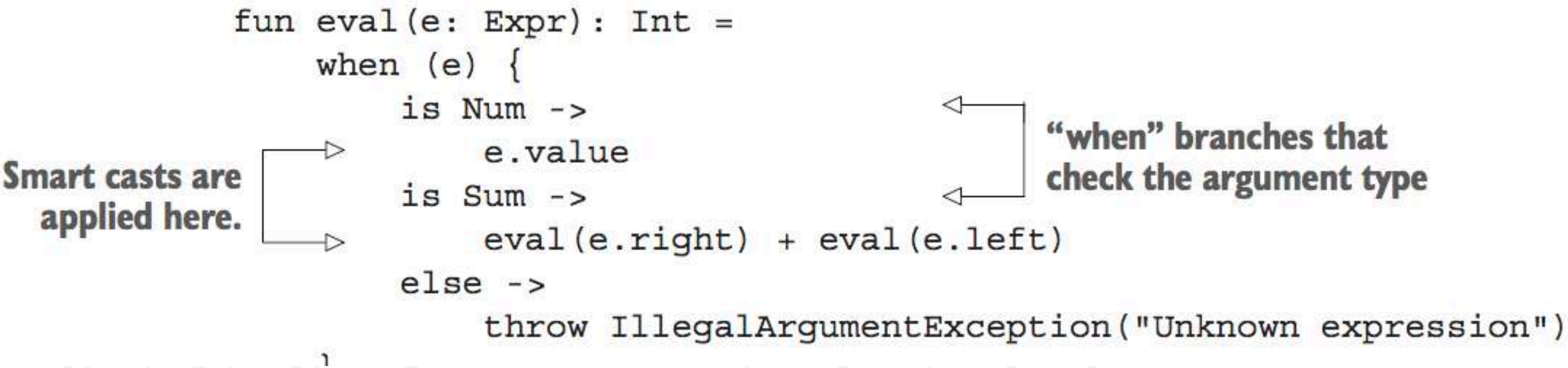
The variable e is smart-cast.

Refactoring: **expresión de retorno**



```
fun eval(e: Expr): Int =
    if (e is Num) {
        e.value
    } else if (e is Sum) {
        eval(e.right) + eval(e.left)
    } else {
        throw IllegalArgumentException("Unknown expression")
    }

>>> println(eval(Sum(Num(1), Num(2))))
3
```

Refactoring: **if** por **when**

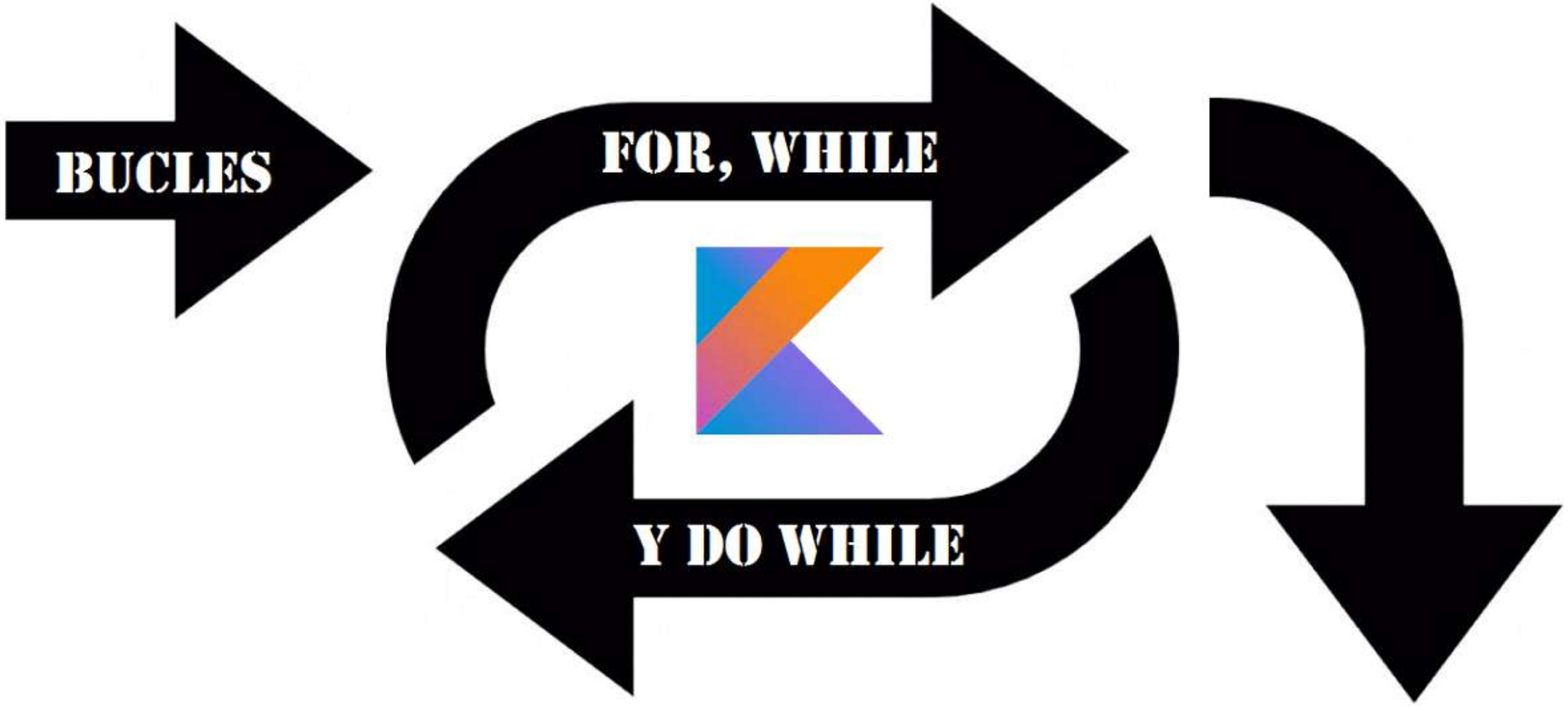


```
fun eval(e: Expr): Int =
    when (e) {
        is Num ->
            e.value
        is Sum ->
            eval(e.right) + eval(e.left)
        else ->
            throw IllegalArgumentException("Unknown expression")
    }
```

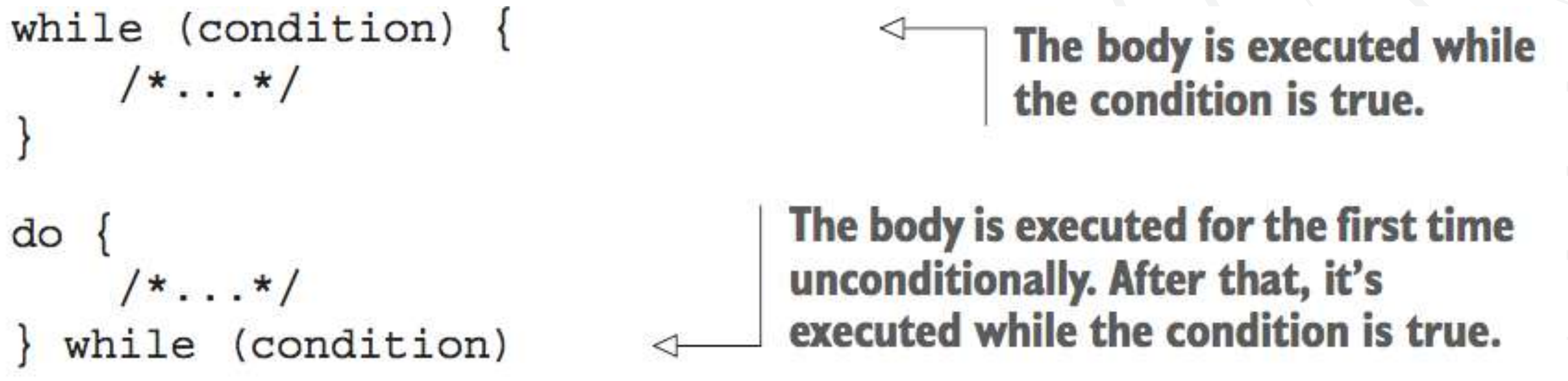
Smart casts are applied here.

“when” branches that check the argument type

Ciclos **while** y **for**



do-while loops



```
while (condition) {
    /*...*/
}
```

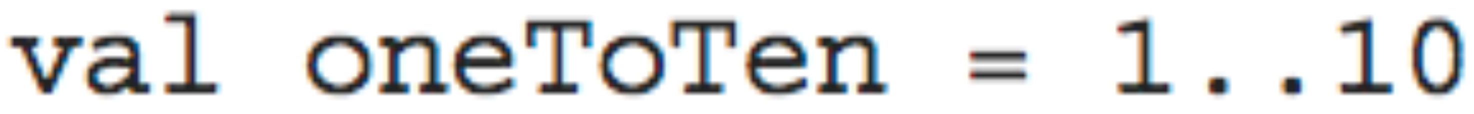
The body is executed while the condition is true.

```
do {
    /*...*/
} while (condition)
```

The body is executed for the first time unconditionally. After that, it's executed while the condition is true.

Rangos y progresiones

- **rango: intervalo entre dos valores**



```
val oneToTen = 1..10
```

Rangos y progresiones

```
for (i in 1.. 100) {

}

for (i in 100 downTo 1 step 2) {

}
```

Iterando sobre mapas

```
val binaryReps = TreeMap<Char, String>()

for (c in 'A'..'F') {
    val binary = Integer.toBinaryString(c.toInt())
    binaryReps[c] = binary
}

for ((letter, binary) in binaryReps) {
    println("$letter = $binary")
}
```

Uses TreeMap so the keys are sorted

Iterates over the characters from A to F using a range of characters

Converts ASCII code to binary

Stores the value in a map by the c key

Iterates over a map, assigning the map key and value to two variables

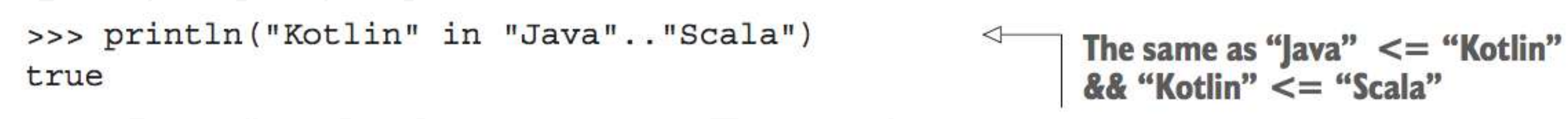
Usando **in** en **when**

```
fun recognize(c: Char) = when (c) {
    in '0'..'9' -> "It's a digit!"
    in 'a'..'z', in 'A'..'Z' -> "It's a letter!"
    else -> "I don't know…"
}
>>> println(recognize('8'))
It's a digit!
```

Checks whether the value is in the range from 0 to 9

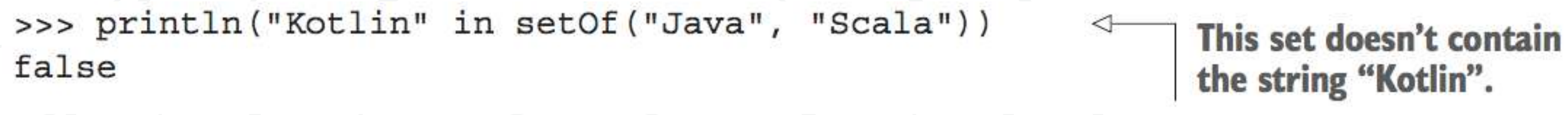
You can combine multiple ranges.

Usando **in** en rangos de Strings



```
>>> println("Kotlin" in "Java".."Scala")
true
```

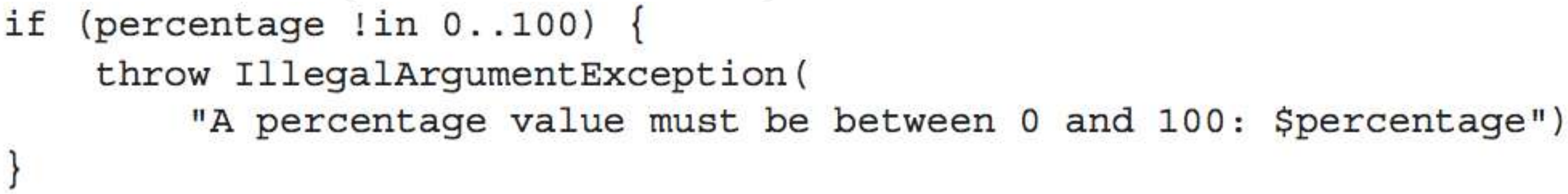
The same as "Java" <= "Kotlin" && "Kotlin" <= "Scala"



```
>>> println("Kotlin" in setOf("Java", "Scala"))
false
```

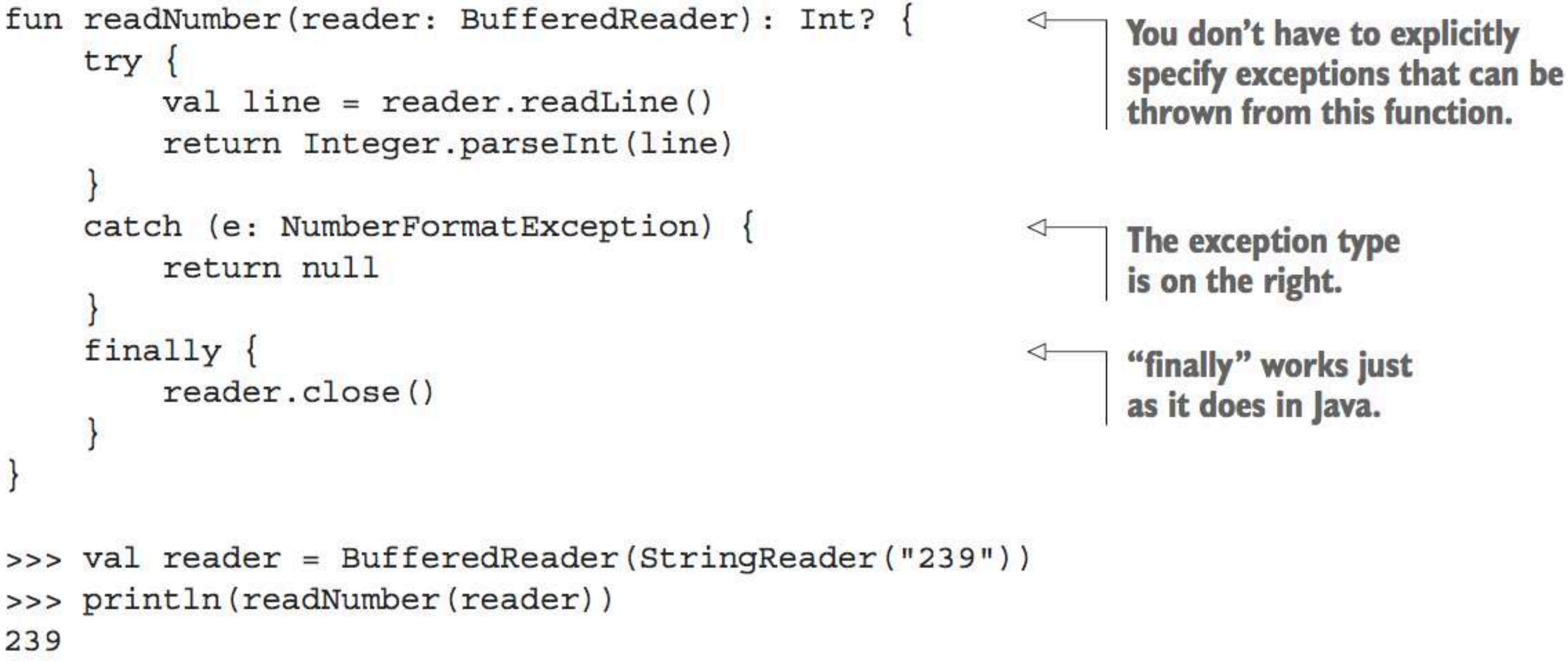
This set doesn't contain the string "Kotlin".

Excepciones en Kotlin



```
if (percentage !in 0..100) {
    throw IllegalArgumentException(
        "A percentage value must be between 0 and 100: $percentage")
}
```

try, catch y finally



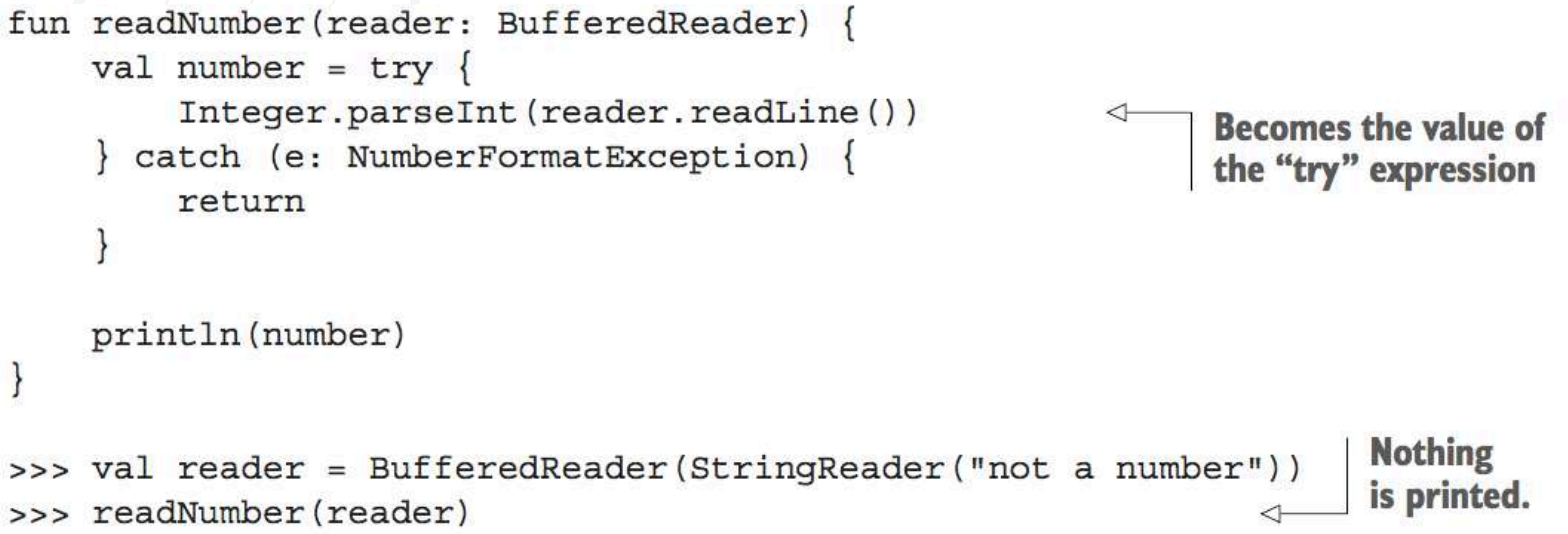
```
fun readNumber(reader: BufferedReader): Int? {
    try {
        val line = reader.readLine()
        return Integer.parseInt(line)
    }
    catch (e: NumberFormatException) {
        return null
    }
    finally {
        reader.close()
    }
}

>>> val reader = BufferedReader(StringReader("239"))
>>> println(readNumber(reader))
239
```

You don’t have to explicitly specify exceptions that can be thrown from this function.

The exception type is on the right.

“finally” works just as it does in Java.

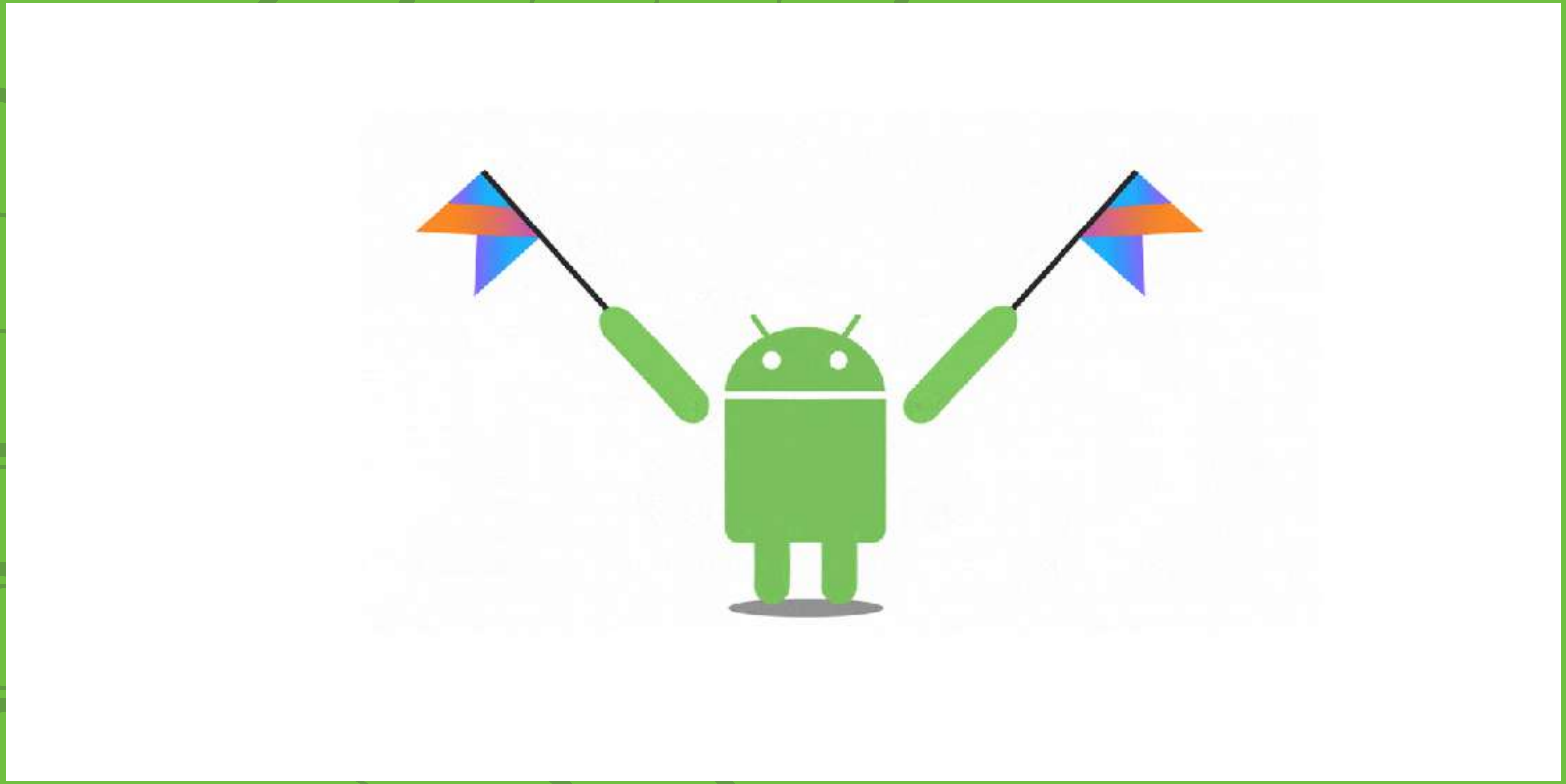
try como expresión



```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        return
    }

    println(number)
}

>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
```

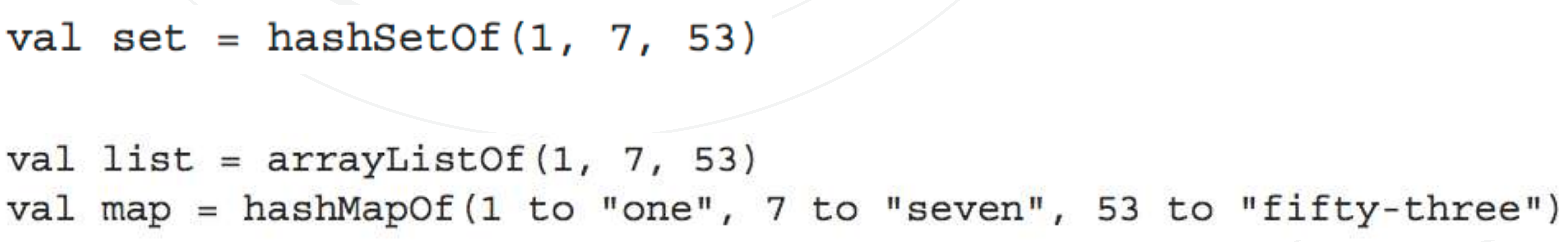
Becomes the value of the "try" expression

Nothing is printed.



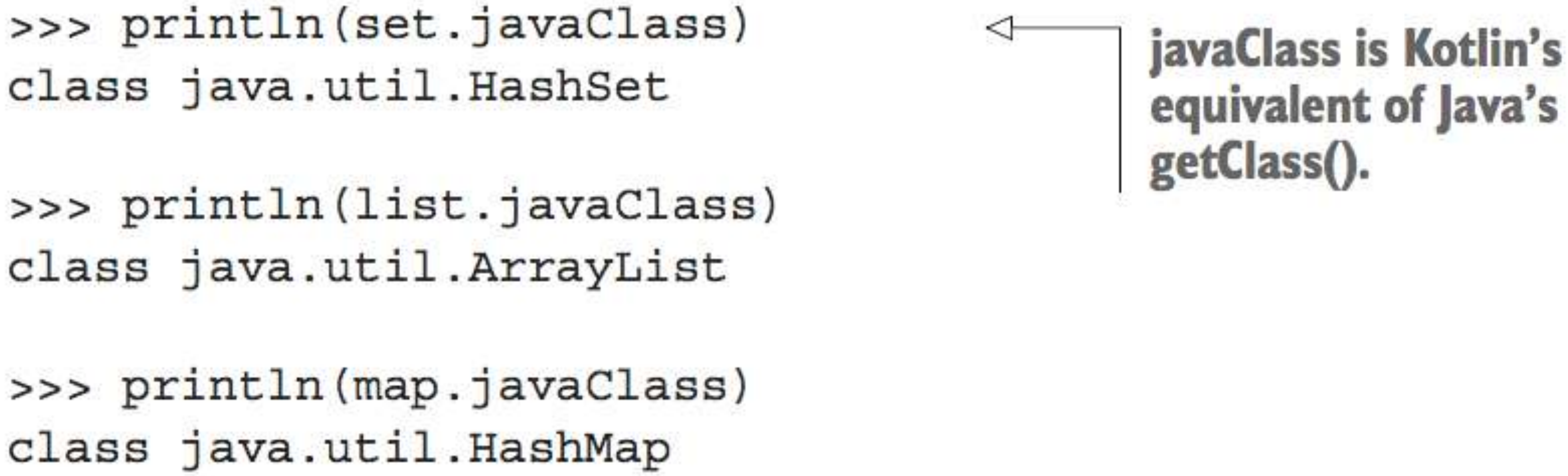
Funciones de Kotlin

Creando colecciones



```
val set = hashSetOf(1, 7, 53)

val list = arrayListOf(1, 7, 53)
val map = hashMapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```

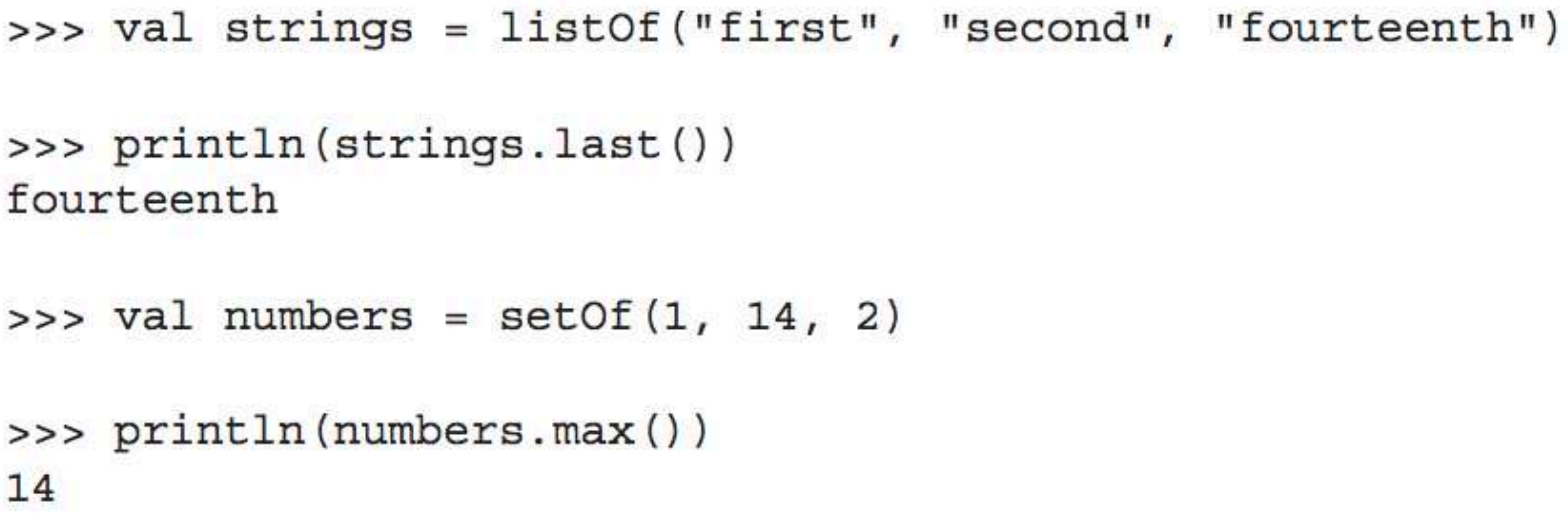
Creando colecciones



```
>>> println(set.javaClass)
class java.util.HashSet

>>> println(list.javaClass)
class java.util.ArrayList

>>> println(map.javaClass)
class java.util.HashMap
```

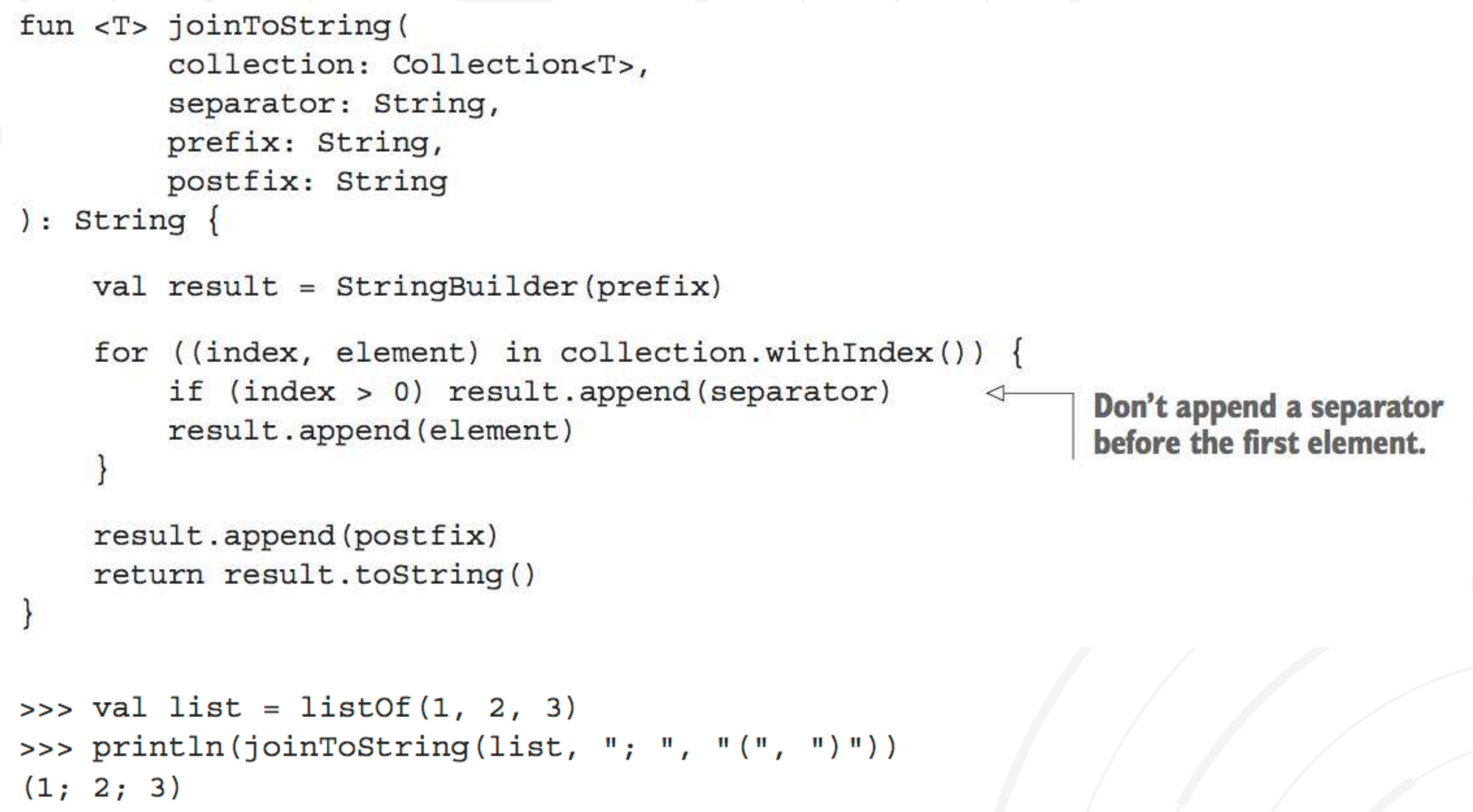
javaClass is Kotlin's equivalent of Java's getClass().

Creando colecciones



```
>>> val strings = listOf("first", "second", "fourteenth")

>>> println(strings.last())
fourteenth

>>> val numbers = setOf(1, 14, 2)

>>> println(numbers.max())
14
```

Llamando funciones

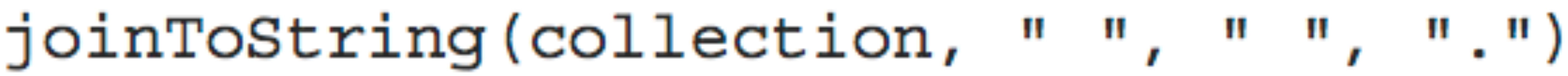


```
fun <T> joinToString(
        collection: Collection<T>,
        separator: String,
        prefix: String,
        postfix: String
): String {

    val result = StringBuilder(prefix)

    for ((index, element) in collection.withIndex()) {
        if (index > 0) result.append(separator)
        result.append(element)
    }

    result.append(postfix)
    return result.toString()
}
```

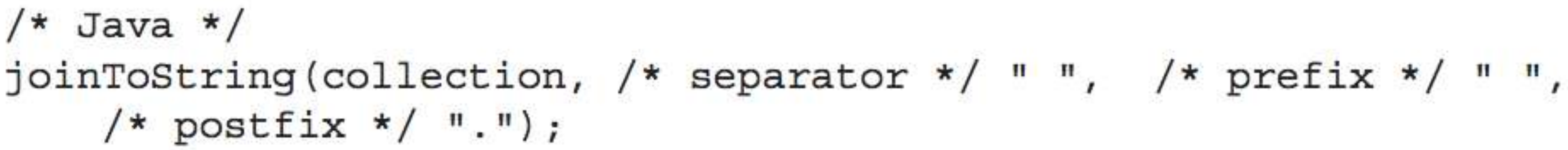
Don’t append a separator before the first element.

```
>>> val list = listOf(1, 2, 3)
>>> println(joinToString(list, "; ", "(", ")"))
(1; 2; 3)
```

Argumentos con nombre



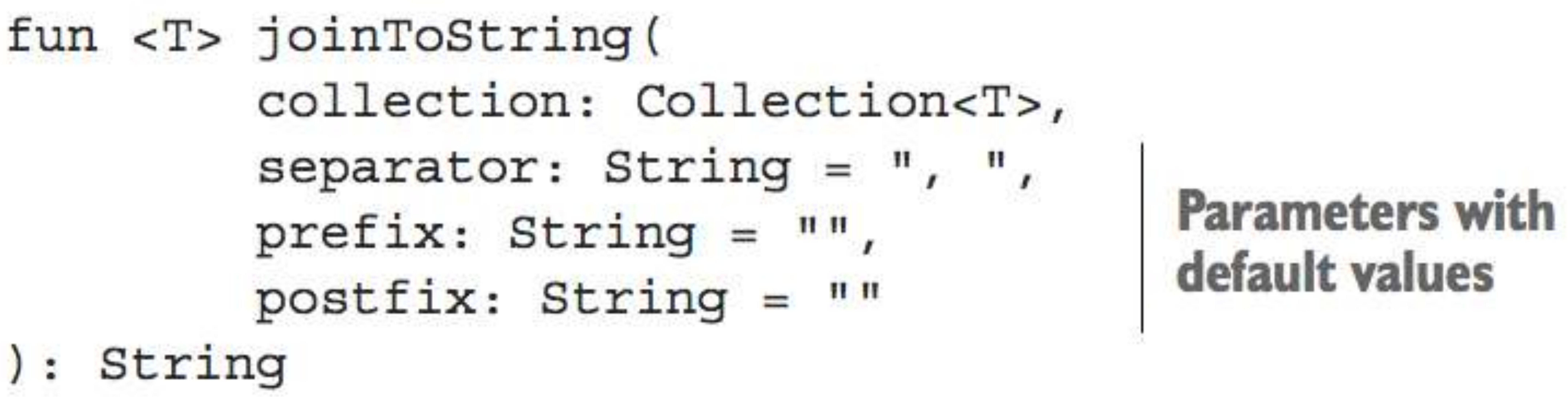
```
joinToString(collection, " ", " ", ".")
```



```
/* Java */
joinToString(collection, /* separator */ " ",  /* prefix */ " ",
    /* postfix */ ".");
```

```
joinToString(collection, separator = " ", prefix = " ", postfix = ".")
```

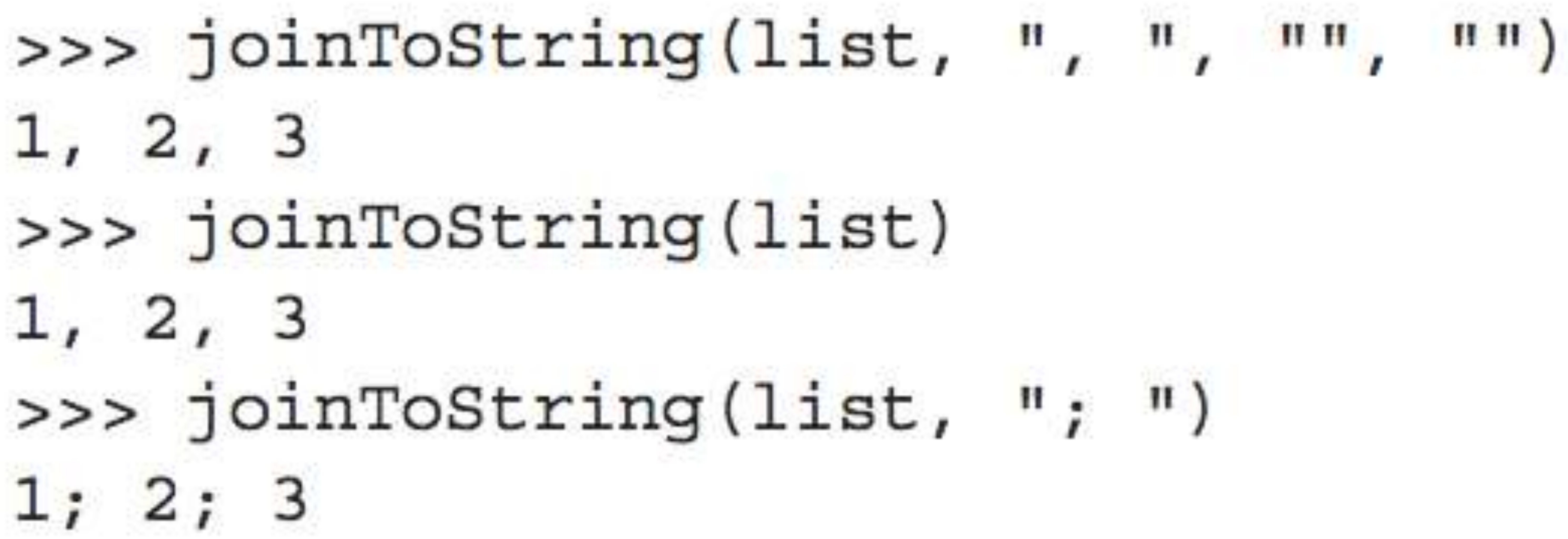
Parámetros con valores por defecto



```
fun <T> joinToString(
        collection: Collection<T>,
        separator: String = ", ",
        prefix: String = "",
        postfix: String = ""
): String
```

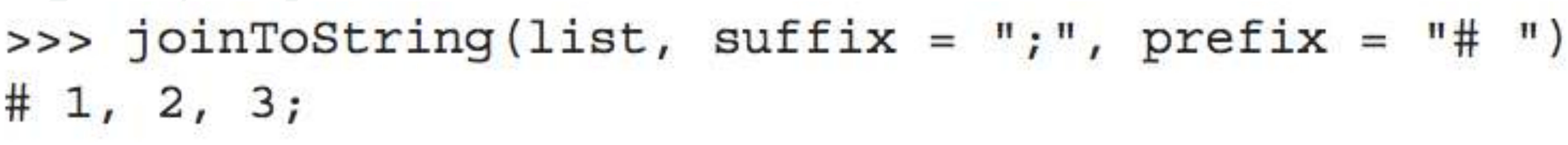
Parameters with default values

Parámetros con valores por defecto



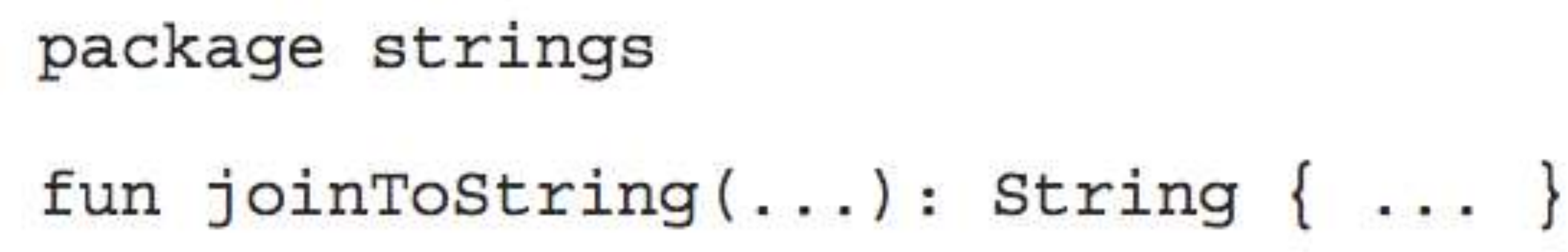
```
>>> joinToString(list, ", ", "", "")
1, 2, 3
>>> joinToString(list)
1, 2, 3
>>> joinToString(list, "; ")
1; 2; 3
```

Parámetros con valores por defecto



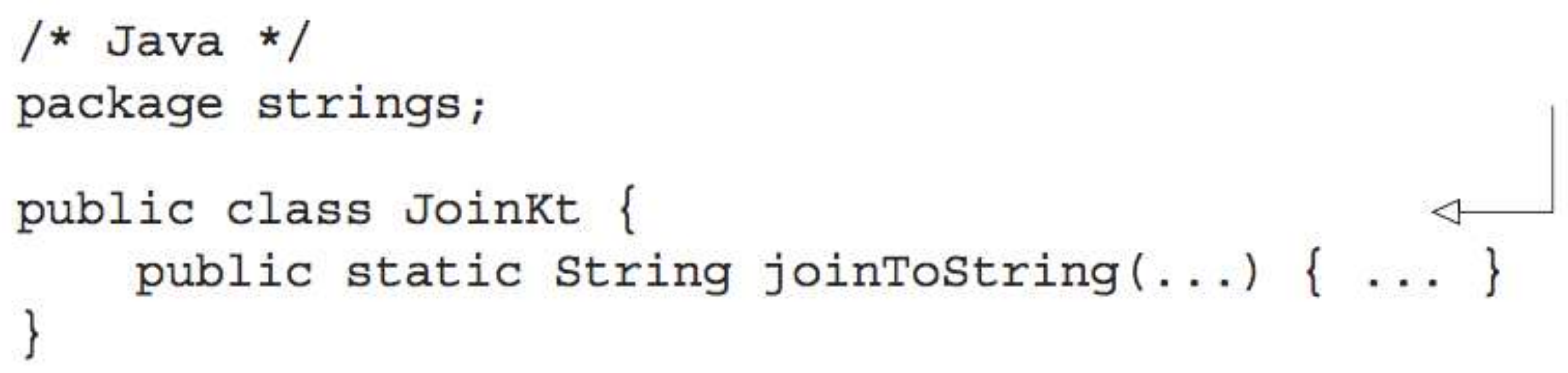
```
>>> joinToString(list, suffix = ";", prefix = "# ")
# 1, 2, 3;
```

Funciones Top-Level

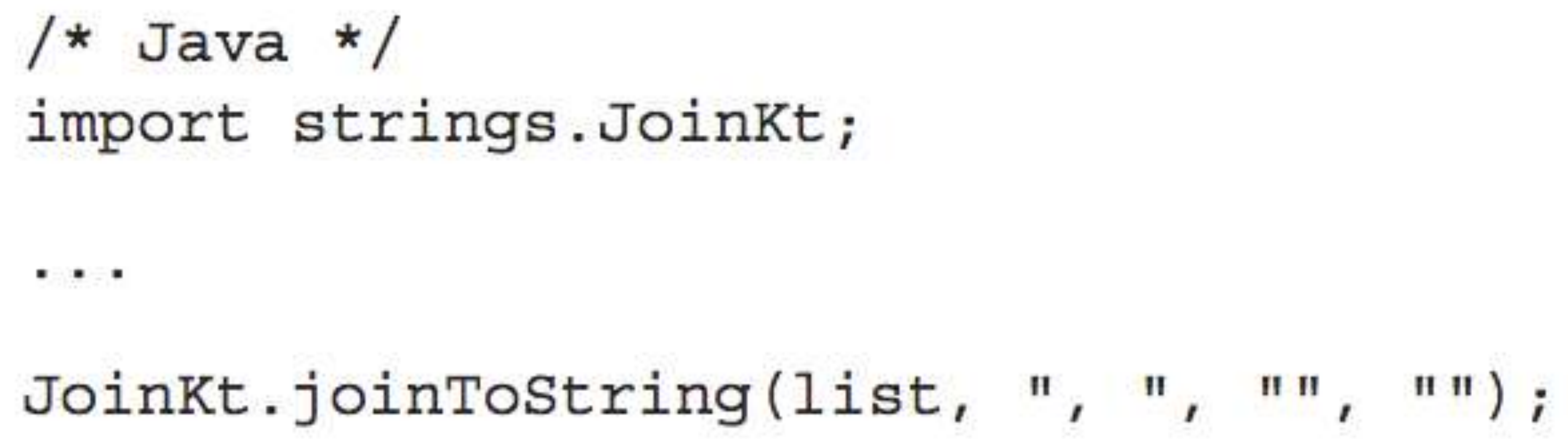


```
package strings

fun joinToString(...): String { ... }
```

Funciones Top-Level

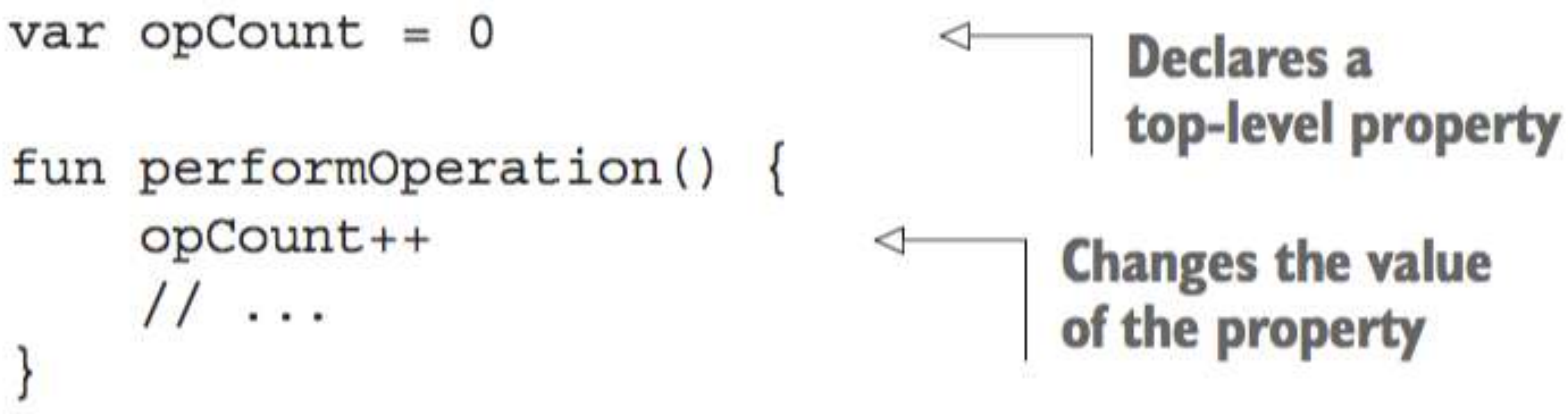


```
/* Java */
package strings;

public class JoinKt {
    public static String joinToString(...) { ... }
}
```



```
/* Java */
import strings.JoinKt;

...

JoinKt.joinToString(list, ", ", "", "");
```

Propiedades Top-Level

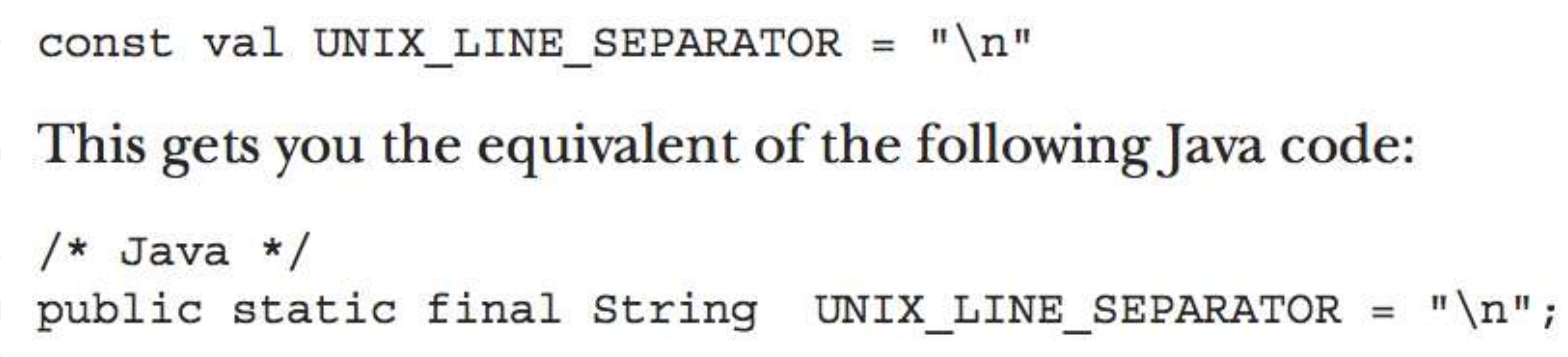


```
var opCount = 0

fun performOperation() {
    opCount++
    // ...
}
```

Declares a top-level property

Changes the value of the property

Propiedades Top-Level

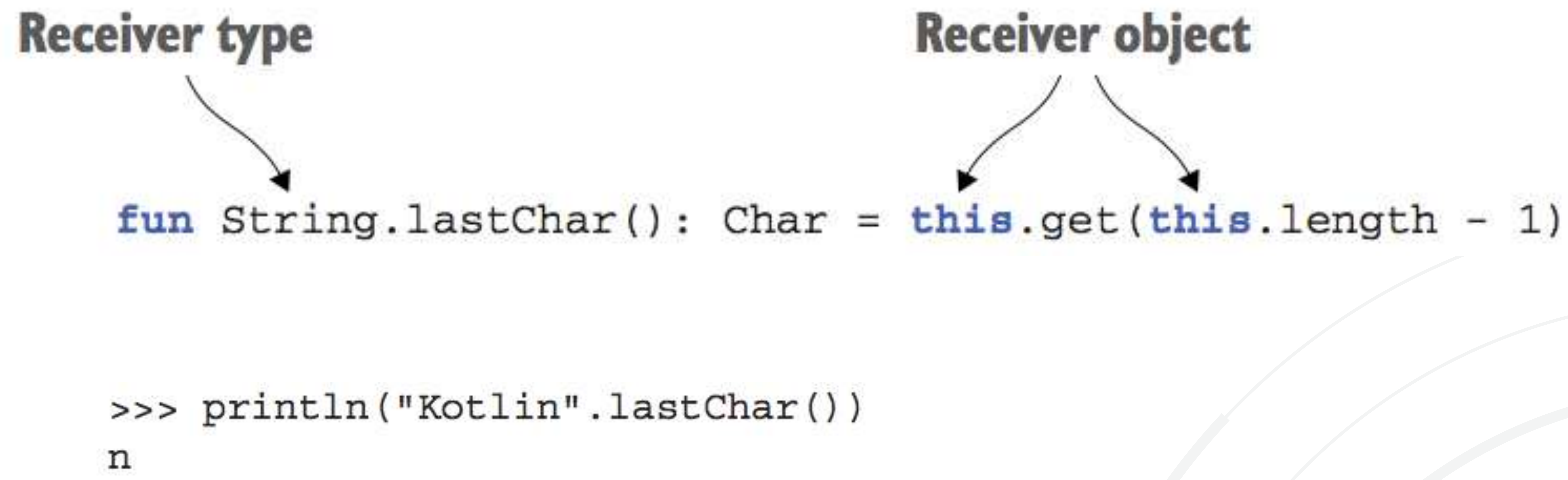


```
const val UNIX_LINE_SEPARATOR = "\n"
```

This gets you the equivalent of the following Java code:

```
/* Java */
public static final String  UNIX_LINE_SEPARATOR = "\n";
```

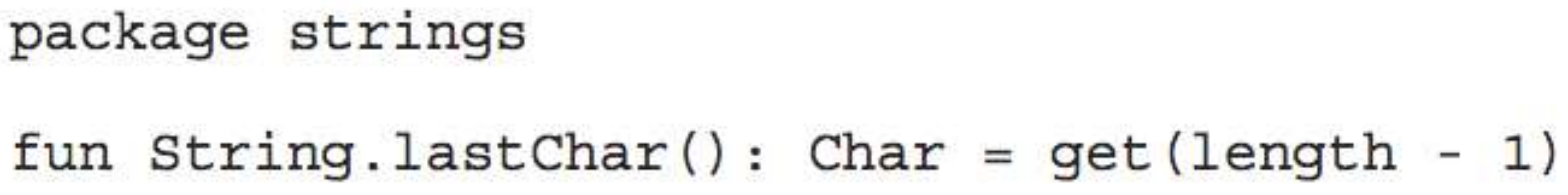
Funciones de extensión



```
fun String.lastChar(): Char = this.get(this.length - 1)
```

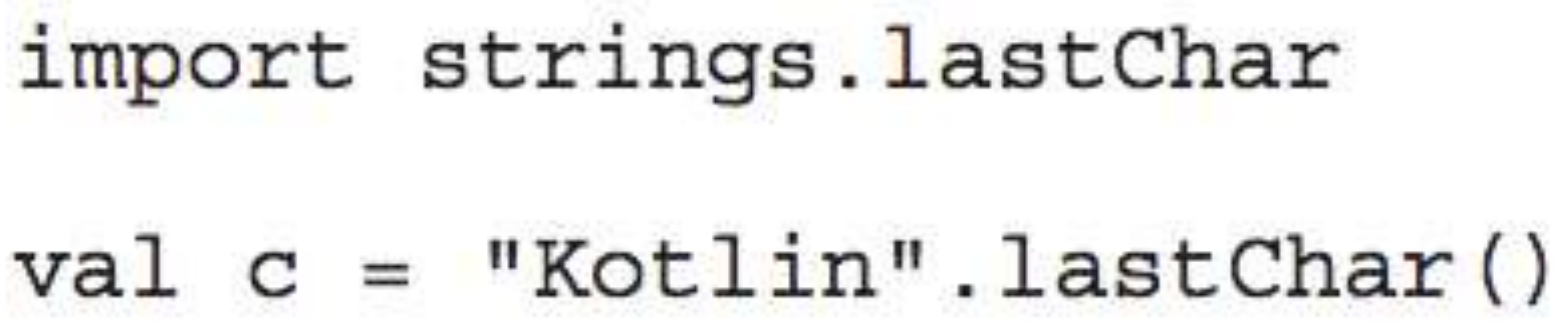
```
>>> println("Kotlin".lastChar())
n
```

Funciones de extensión

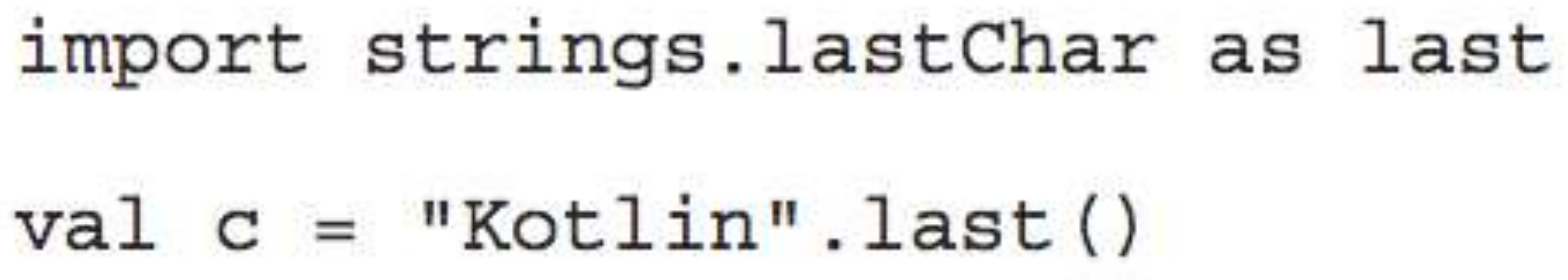


```
package strings

fun String.lastChar(): Char = get(length - 1)
```

Imports y Funciones de extensión

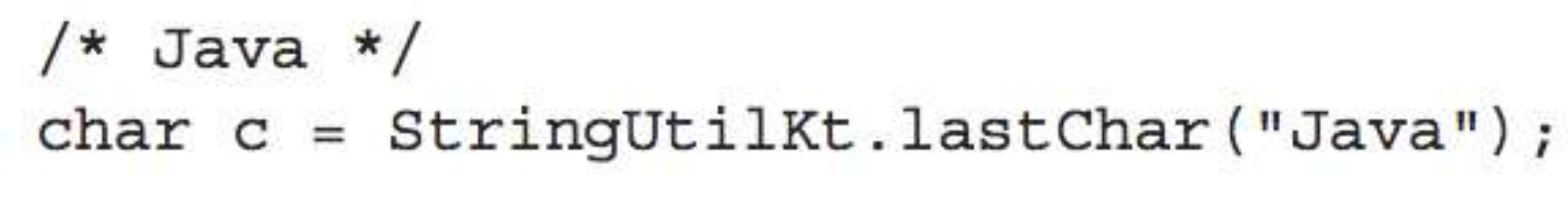


```
import strings.lastChar

val c = "Kotlin".lastChar()
```



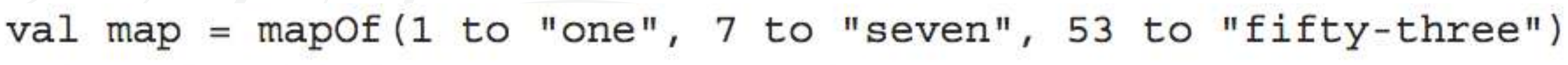
```
import strings.lastChar as last

val c = "Kotlin".last()
```

Llamando a las Funciones de extensión desde java

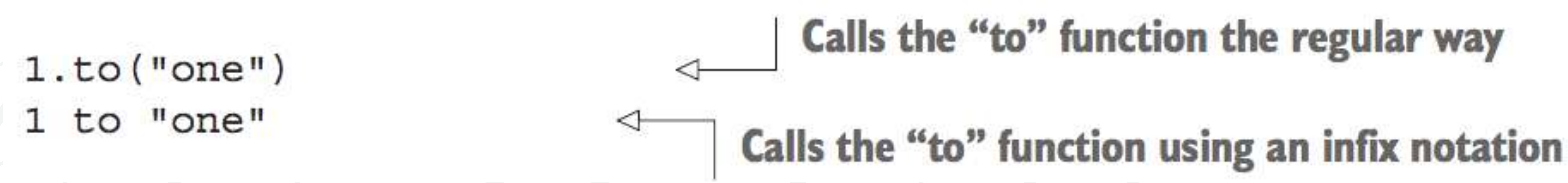


```
/* Java */
char c = StringUtilKt.lastChar("Java");
```

Pares: infix calls y des-estructurando declaraciones



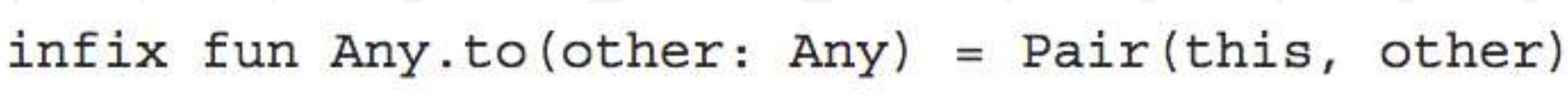
```
val map = mapOf(1 to "one", 7 to "seven", 53 to "fifty-three")
```



```
1.to("one")
1 to "one"
```

Calls the "to" function the regular way

Calls the "to" function using an infix notation



```
infix fun Any.to(other: Any) = Pair(this, other)
```