Kotlin

Basic Syntax

Defining packages

Package specification should be at the top of the source file:

```
package my.demo

import java.util.*

// ...
```

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

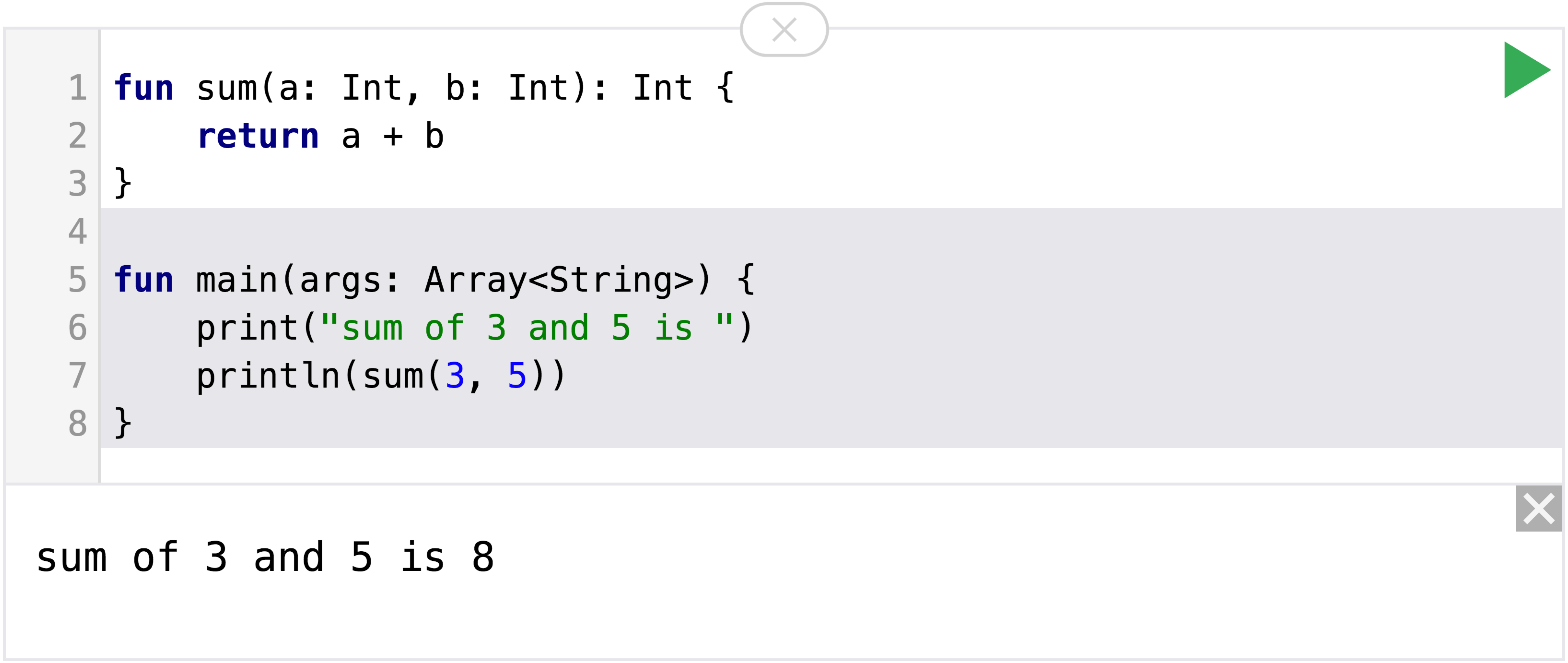
Defining functions

Function having two `Int` parameters with `Int` return type:

```kotlin
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Defining functions

Function having two `Int` parameters with `Int` return type:



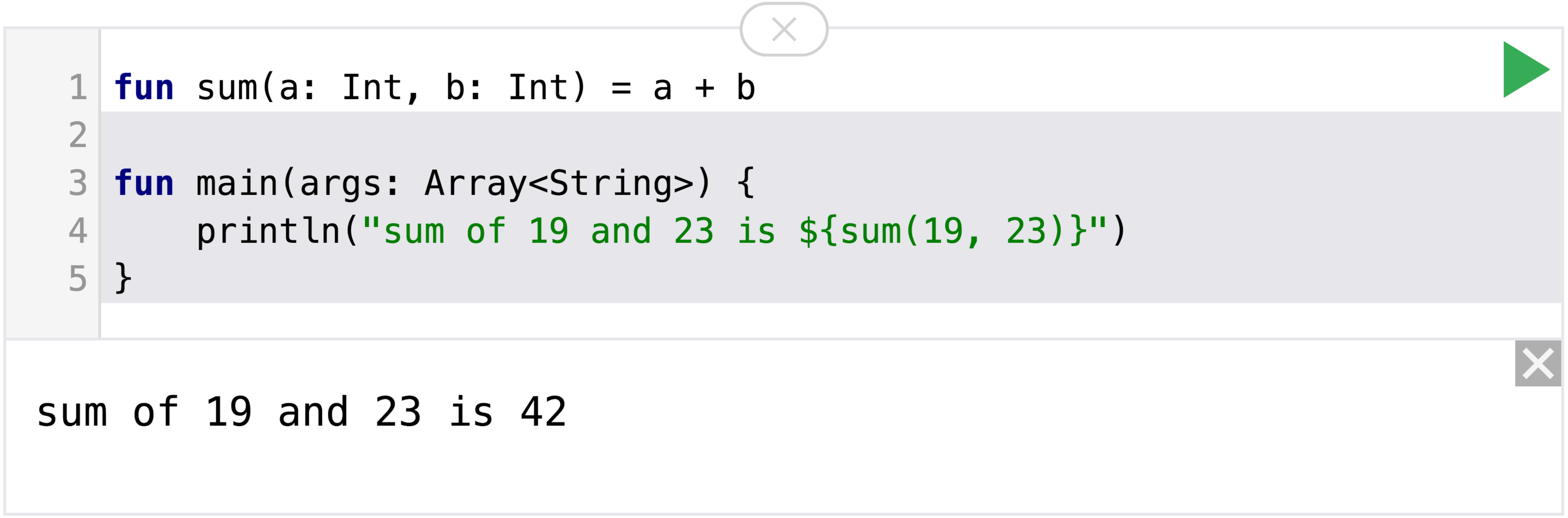
```kotlin
fun sum(a: Int, b: Int): Int {
    return a + b
}

fun main(args: Array<String>) {
    print("sum of 3 and 5 is ")
    println(sum(3, 5))
}
```

```
sum of 3 and 5 is 8
```

Function with an expression body and inferred return type:

```kotlin
fun sum(a: Int, b: Int) = a + b
```

Function with an expression body and inferred return type:



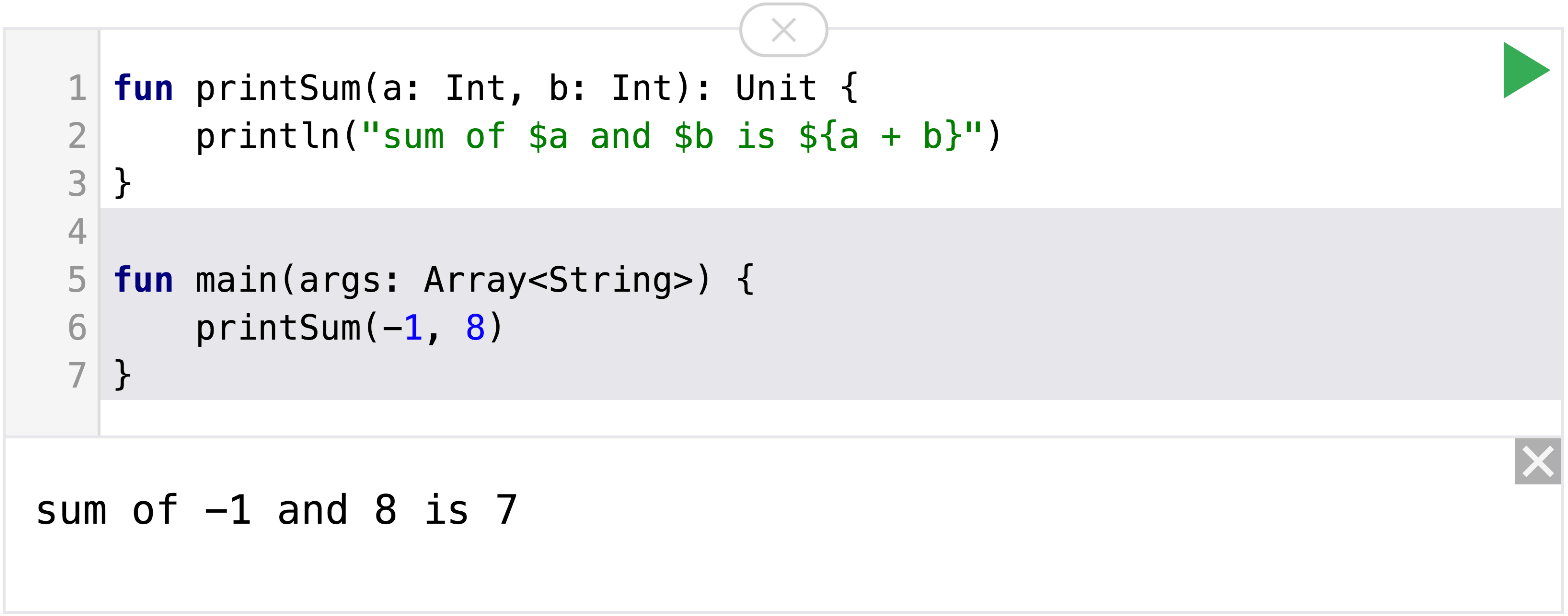
```kotlin
fun sum(a: Int, b: Int) = a + b

fun main(args: Array<String>) {
    println("sum of 19 and 23 is ${sum(19, 23)}")
}
```

```
sum of 19 and 23 is 42
```

Function returning no meaningful value:

```kotlin
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

Function returning no meaningful value:



```kotlin
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}

fun main(args: Array<String>) {
    printSum(-1, 8)
}
```

```
sum of -1 and 8 is 7
```

`Unit` return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

Defining variables

Assign-once (read-only) local variable:

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

Target platform: JVM   Running on kotlin v. 1.2.70

Mutable variable:

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

Top-level variables:

```kotlin
val PI = 3.14
var x = 0

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

Comments

Just like Java and JavaScript, Kotlin supports end-of-line and block comments.

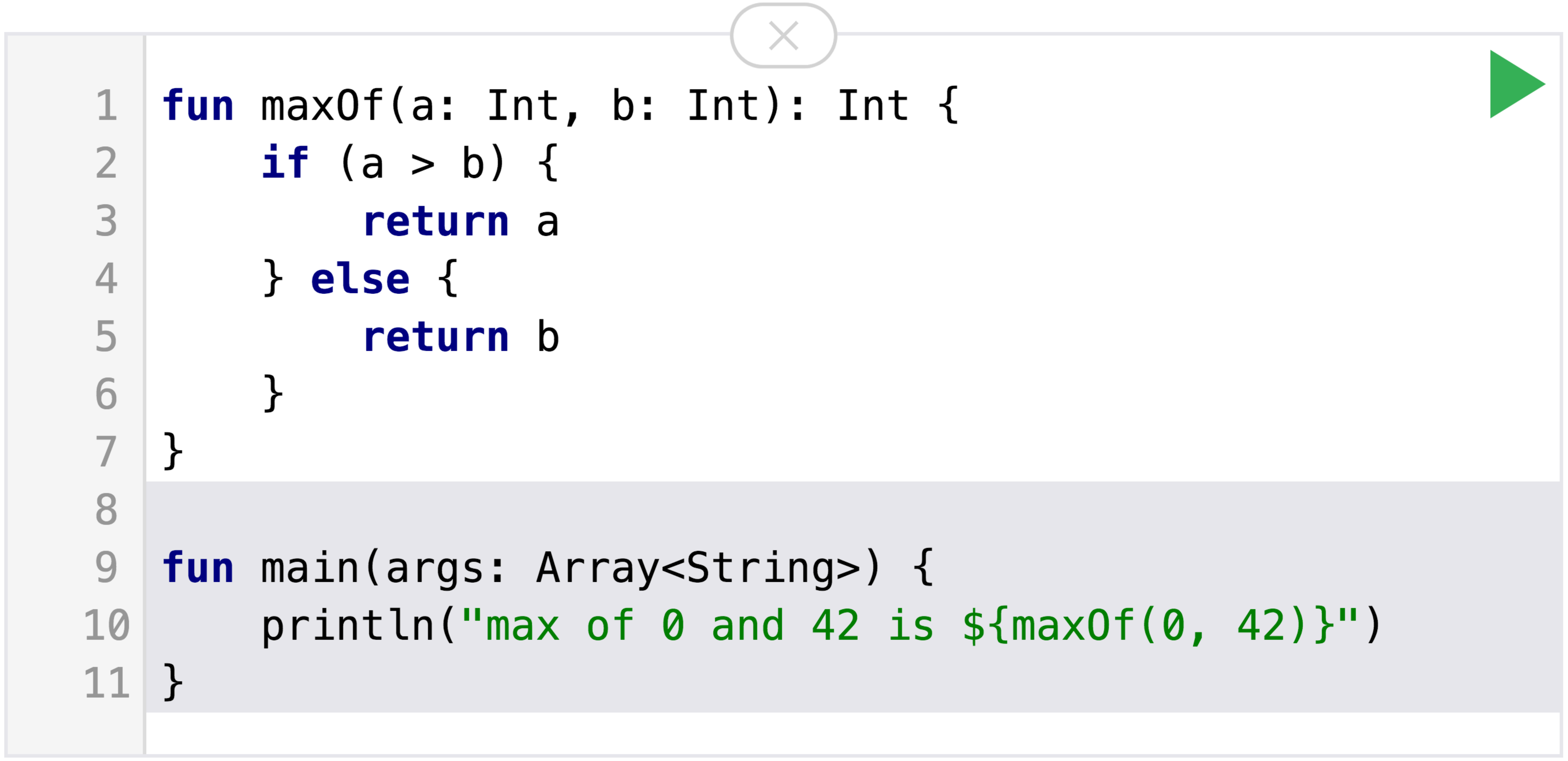
```
// This is an end-of-line comment

/* This is a block comment
   on multiple lines. */
```

Unlike Java, block comments in Kotlin can be nested.

Using string templates

```
var a = 1
// simple name in template:
val s1 = "a is $a"

a = 2
// arbitrary expression in template:
val s2 = "${s1.replace("is", "was")}, but now is $a"
```

Using conditional expressions



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

fun main(args: Array<String>) {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

Using `if` as an expression:

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

Using nullable values and checking for `null`

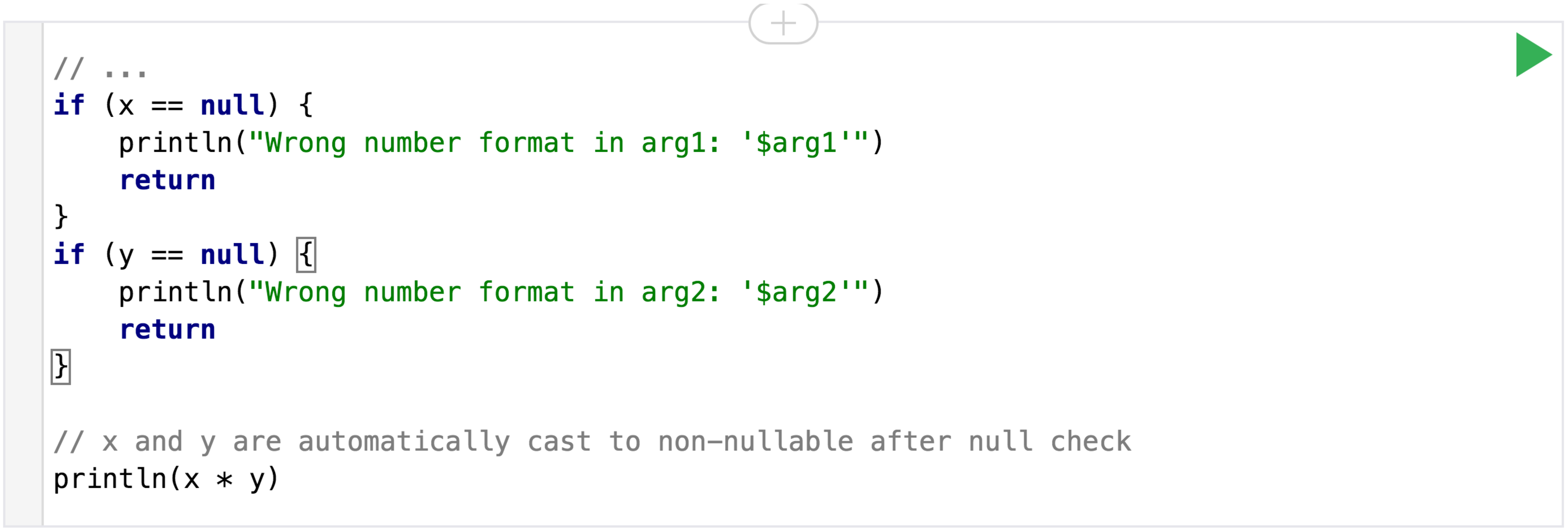
A reference must be explicitly marked as nullable when `null` value is possible.

Return `null` if `str` does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Use a function returning nullable value:

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("either '$arg1' or '$arg2' is not a number")
    }
}
```

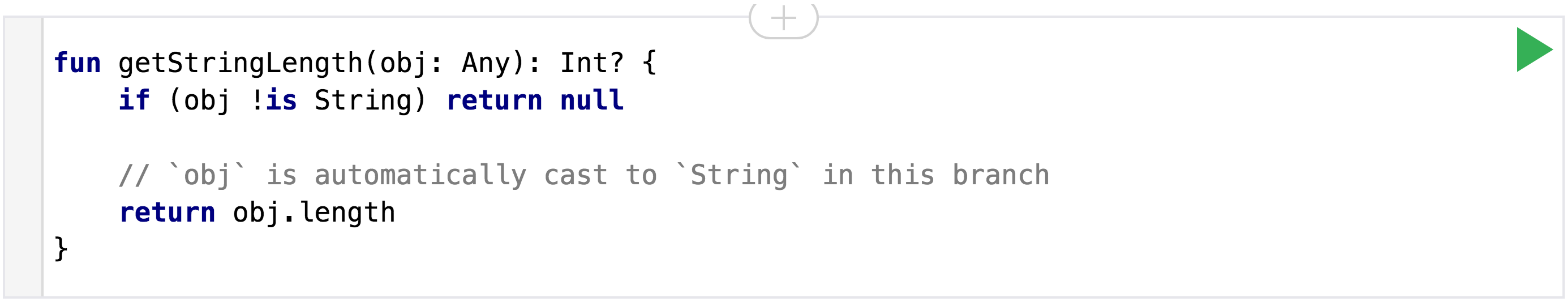


```kotlin
// ...
if (x == null) {
    println("Wrong number format in arg1: '$arg1'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '$arg2'")
    return
}

// x and y are automatically cast to non-nullable after null check
println(x * y)
```

Using type checks and automatic casts

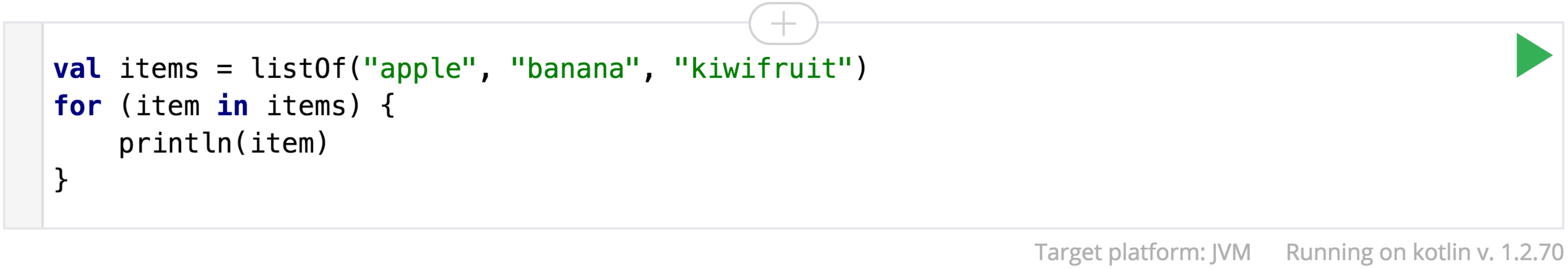
The `is` operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

```kotlin
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}
```



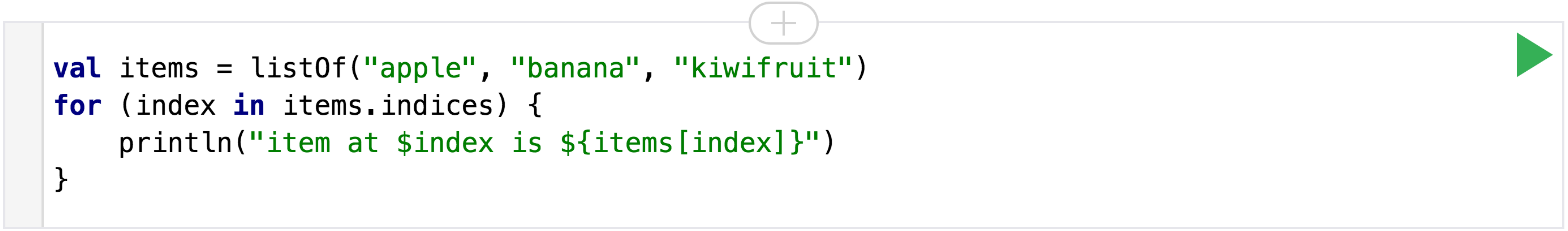
```kotlin
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` is automatically cast to `String` in this branch
    return obj.length
}
```

Using a `for` loop



```kotlin
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

or



```kotlin
val items = listOf("apple", "banana", "kiwifruit")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}
```

Using a `while` loop

```kotlin
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}
```

Using when expression

```
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
```

Using ranges

Check if a number is within a range using `in` operator:

```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

Check if a number is out of range:

```kotlin
val list = listOf("a", "b", "c")

if (-1 !in 0..list.lastIndex) {
    println("-1 is out of range")
}
if (list.size !in list.indices) {
    println("list size is out of valid list indices range too")
}
```

Using collections

Iterating over a collection:

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

Target platform: JVM   Running on kotlin v. 1.2.70

Checking if a collection contains an object using `in` operator:

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

Checking if a collection contains an object using `in` operator:

```kotlin
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

Using lambda expressions to filter and map collections:

```kotlin
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
fruits
  .filter { it.startsWith("a") }
  .sortedBy { it }
  .map { it.toUpperCase() }
  .forEach { println(it) }
```

Creating basic classes and their instances:

```
val rectangle = Rectangle(5.0, 2.0) //no 'new' keyword required
val triangle = Triangle(3.0, 4.0, 5.0)
```