# A program to calculate the factorial of a given number.

fun factorial(num:Int):Int{ var result: Int = 1

for(i in 2..num){ result \*= i

}

return result

}

fun main(){

val num = 5

val factFun = *factorial*(num) *println*(factFun)

}

# A program that uses map and filter functions on a list of strings.

**Map Function**: The map function applies a given transformation or operation to each element of a collection, creating a new collection with the results. It takes a function as an argument and applies it to each element, mapping it to a new value. The resulting collection has the same size as the original collection.

**Filter Function**: The filter function selects elements from a collection based on a given condition or predicate. It takes the function that evaluates the condition for each element. Only the elements that satisfy the condition are included in the resulting collection.

fun main() {

val fruits = *listOf*("apple", "banana", "cherry", "date", "elderberry")

// Example using map function

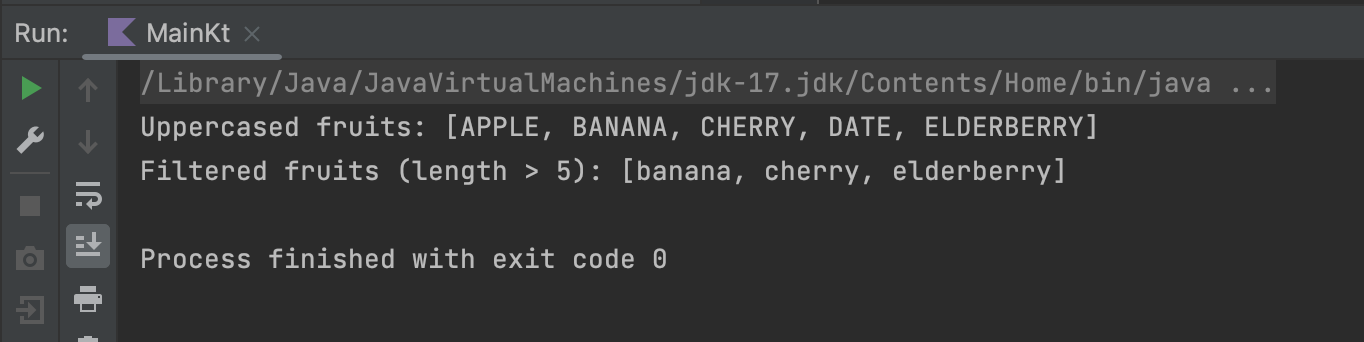
val uppercasedFruits = fruits.*map* **{ it**.*toUpperCase*() **}**

*println*("Uppercased fruits: $uppercasedFruits")

// Example using filter function

val filteredFruits = fruits.*filter* **{ it**.length > 5 **}**

*println*("Filtered fruits (length > 5): $filteredFruits")

}

# A program that checks if a user-inputted string is a palindrome.

fun checkPalindrome(input: String): Boolean { val reversed = input.reversed()

return input == reversed

}

fun main() {

println("Enter a string:") val input = readLine()

if (input != null) {

val sanitizedInput = input.toLowerCase().replace("\\W".toRegex(), "") val isPalindrome = checkPalindrome(sanitizedInput)

if (isPalindrome) {

println("The input string is a palindrome.")

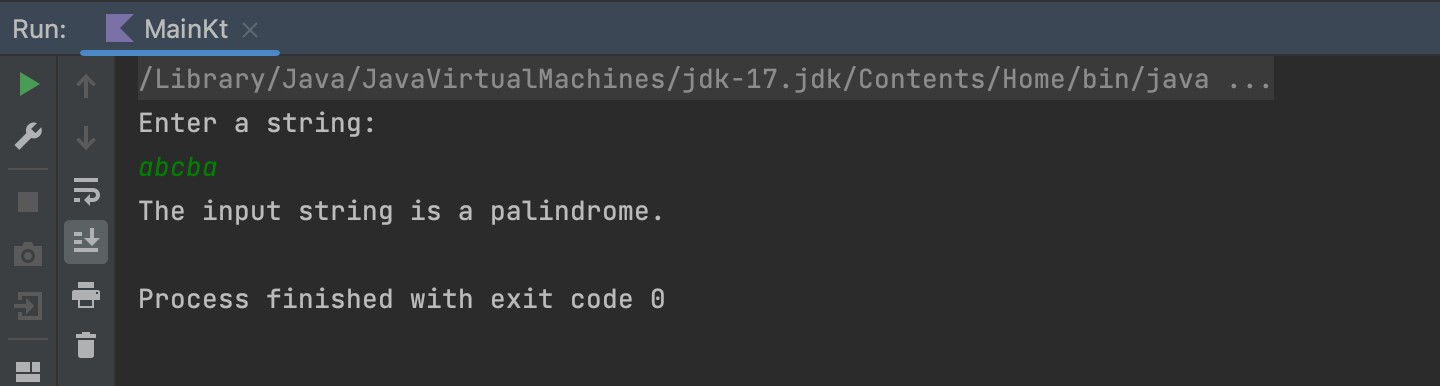
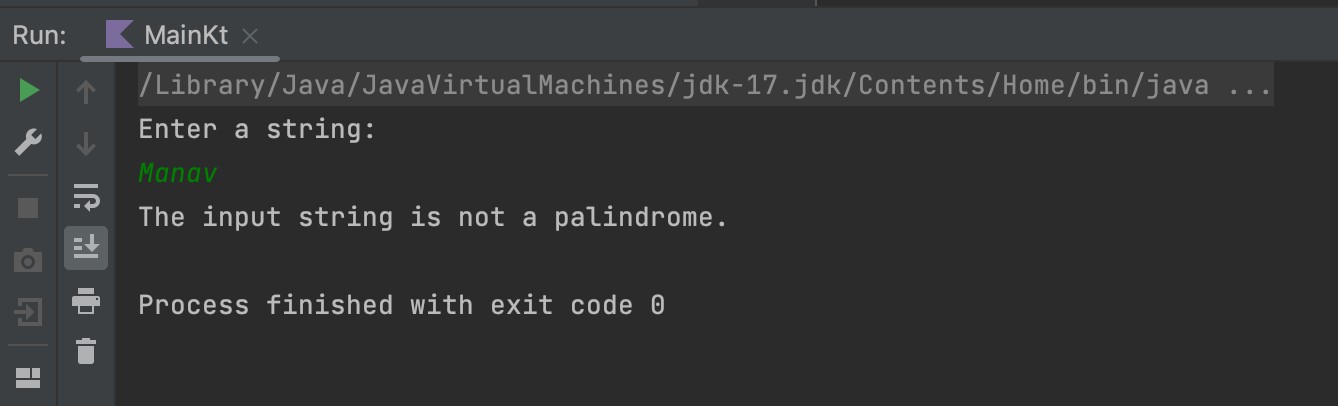
} else {

println("The input string is not a palindrome.")

}

}

}



# An extension function for the List<Int> class that calculates and returns the average of the list elements.

fun List<Int>.calculateAverage(): Double { if (isEmpty()) {

return 0.0

}

val sum = *sum*()

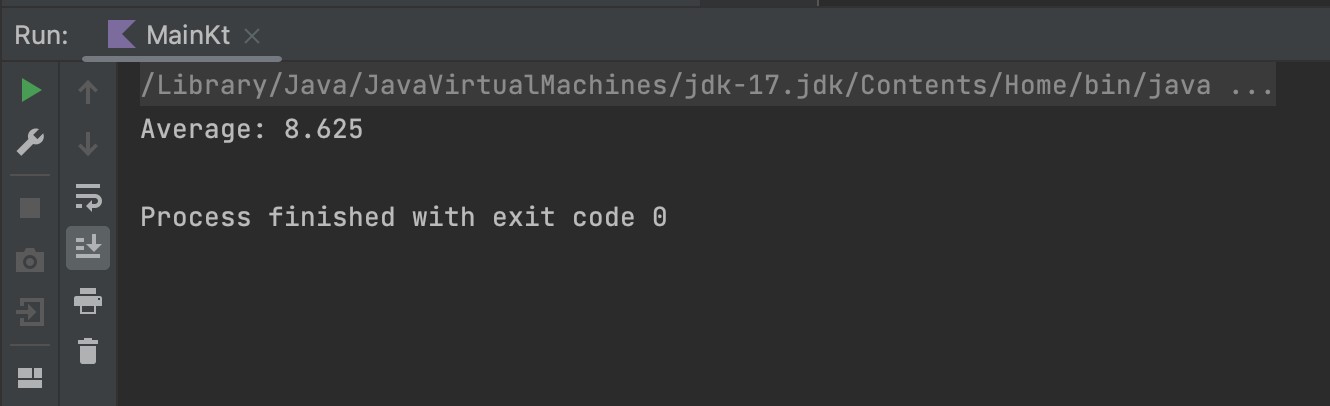
return sum.toDouble() / size

}

fun main() {

val numbers = *listOf*(2, 4, 6, 8, 10, 11, 13, 15) val average = numbers.*calculateAverage*() *println*("Average: $average")

}



# A program that demonstrates the use of let, with, run, apply, and also functions in Kotlin.

fun main(){

val original = " Golden State Warriors!!" var r = original.*let***{it**.*trim*()**}**

.*also***{***println*("Before let : $original, After let : $**it**")**}**

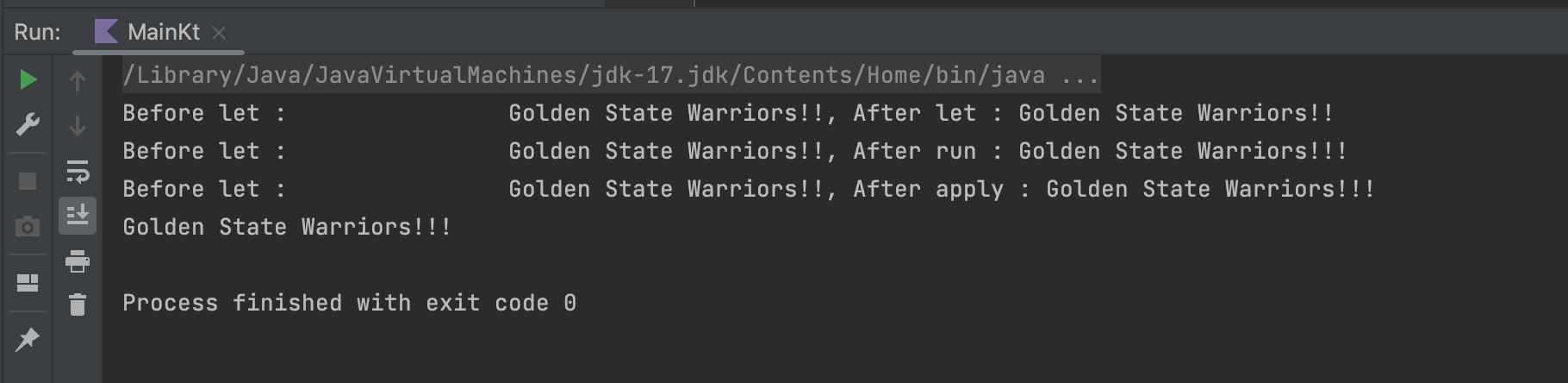
.*run***{**this + "!"**}**

.*also***{***println*("Before let : $original, After run : $**it**")**}**

.*apply***{***println*("Before let : $original, After apply : $this")**}**

*println*(r)

}



* let: The let function allows you to perform operations on an object within a lambda expression and returns the result of the lambda. It's useful for executing a chain of operations on an object and returning a result.
* with: The with function allows you to perform operations on an object within a lambda expression without the need for the it keyword. It's useful for accessing multiple properties or functions of an object in a concise way.
* run: The run function is similar to let, but it operates on the object itself (this). It allows you to perform operations on an object within a lambda expression and returns the result of the lambda. It's useful for executing a series of operations on an object.
* apply: The apply function allows you to configure properties or perform operations on an object within a lambda expression. It returns the object itself after applying the changes. It's useful for initializing an object or modifying its properties.
* also: The also function is similar to apply, but it operates on the object itself (this). It allows you to perform additional operations on an object within a lambda expression and returns the object itself. It's useful for performing side effects while working with an object.
* let, and run both does the same work and it works with the keyword "it".
* apply does the same work as let but works with the keyword "this".
* with does the same work as let and apply its just that it is applied on data classes.

# A program that implements a Student class with properties: name, age, and grades.

class Student(val name: String, val age: Int, val grades: List<Double>) { fun calculateAverageGrade(): Double {

if (grades.isEmpty()) { return 0.0

}

val sum = grades.*sum*() return sum / grades.size

}

}

fun main() {

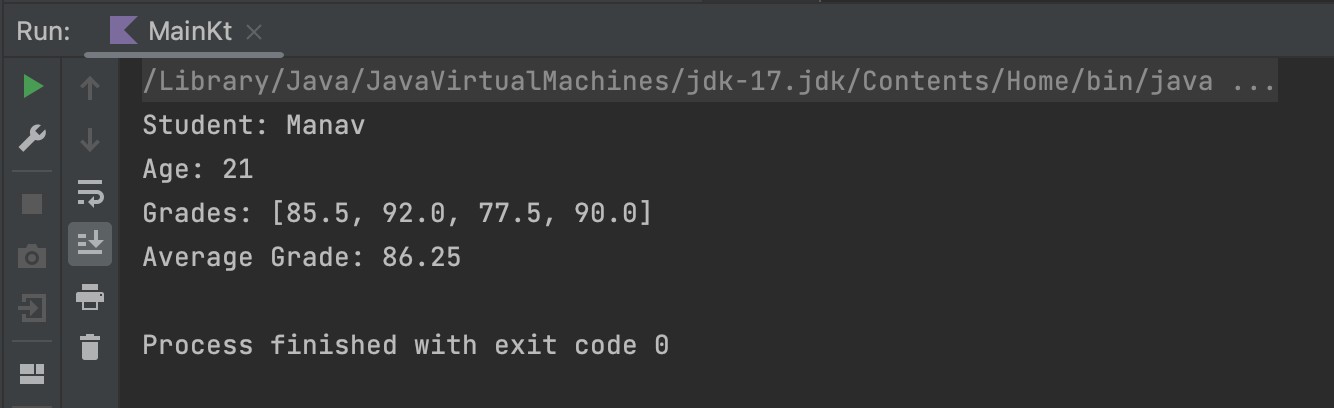
val student = Student("Manav", 21, *listOf*(85.5, 92.0, 77.5, 90.0))

*println*("Student: ${student.name}") *println*("Age: ${student.age}") *println*("Grades: ${student.grades}")

val averageGrade = student.calculateAverageGrade()

*println*("Average Grade: $averageGrade")

}



# A program that converts the Student class into a data class.

In this updated program, we have converted the Student class into a data class by simply adding the data keyword before the class definition. With this change, the Student class automatically provides the implementation of common functions like equals(), hashCode(), toString(), and copy() based on the class properties.

data class Student(val name: String, val age: Int, val grades: List<Double>) { fun calculateAverageGrade(): Double {

if (grades.isEmpty()) { return 0.0

}

val sum = grades.*sum*() return sum / grades.size

}

}

# A program that uses lambda expressions and higher-order functions to find the maximum value in a list of integers.

fun findMax(numbers: List<Int>): Int? { return numbers.*maxOrNull*()

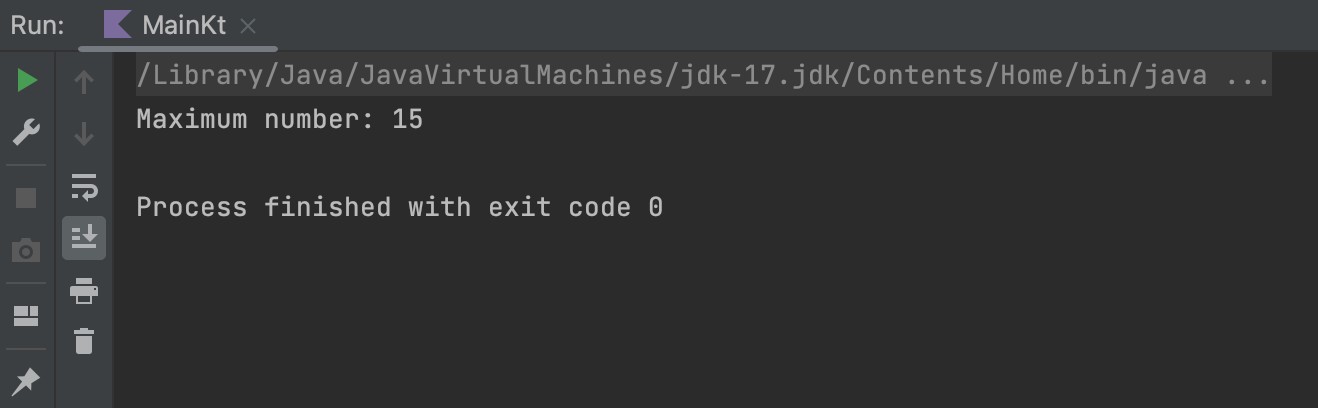
}

fun main() {

val numbers = *listOf*(10, 5, 8, 15, 3, 12) val maxNumber = *findMax*(numbers)

*println*("Maximum number: $maxNumber")

}



In this program, we define a function *findMax* that takes a list of integers (numbers) as a parameter. The findMax function uses the *maxOrNull* higher-order function provided by the Kotlin standard library to find the maximum value in the list. The maxOrNull function returns the maximum value or null if the list is empty.

# A program that implements a Shape interface with two classes, Circle and Rectangle, that implement the Shape interface.

interface Shape { fun area(): Double

}

class Circle(private val radius: Double) : Shape { override fun area(): Double {

return Math.*PI* \* radius \* radius

}

}

class Rectangle(private val width: Double, private val height: Double) : Shape { override fun area(): Double {

return width \* height

}

}

fun main() {

val circle = Circle(3.0)

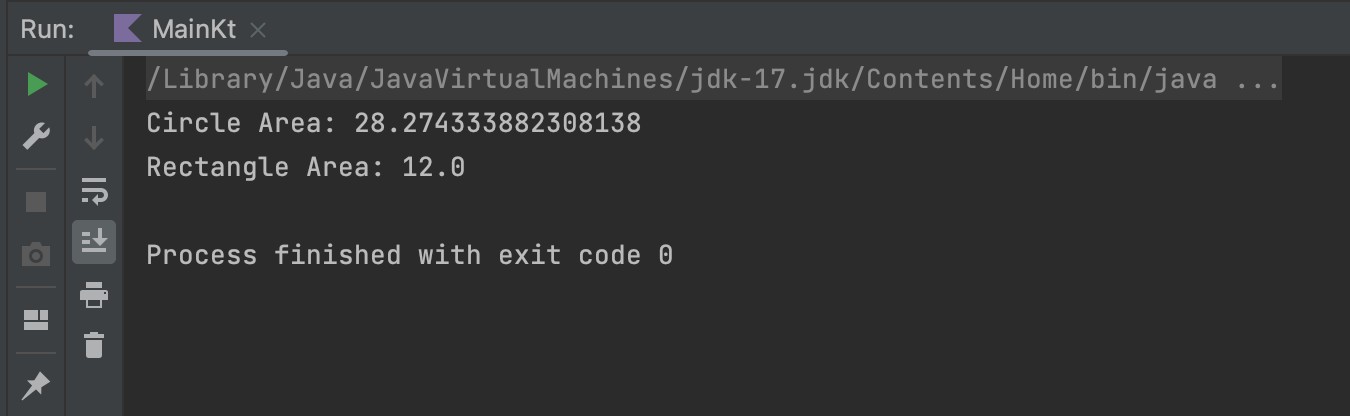
val circleArea = circle.area()

*println*("Circle Area: $circleArea")

val rectangle = Rectangle(3.0, 4.0) val rectangleArea = rectangle.area()

*println*("Rectangle Area: $rectangleArea")

}



# A program to read a text file, count the frequency of each word, and save the results in a new text file.

import java.io.File fun main() {

val inputFile = File("/Users/manav/IdeaProjects/Kotlin\_Lab/src/main/kotlin/

test.txt")

val outputFile = File("/Users/manav/IdeaProjects/Kotlin\_Lab/src/main/kotlin/ output.txt")

val wordCountMap = *mutableMapOf*<String, Int>() inputFile.*forEachLine* **{** line **->**

val words = line.*trim*().*split*("\\s+".*toRegex*())

for (word in words) {

val count = wordCountMap.getOrDefault(word, 0) wordCountMap[word] = count + 1

}

**}**

outputFile.*printWriter*().*use* **{** writer **->**

for ((word, count) in wordCountMap) { writer.println("$word: $count")

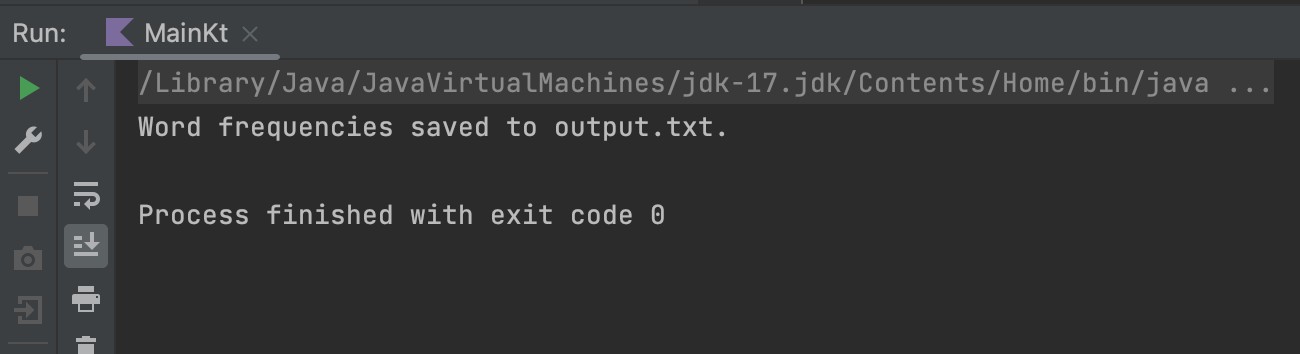
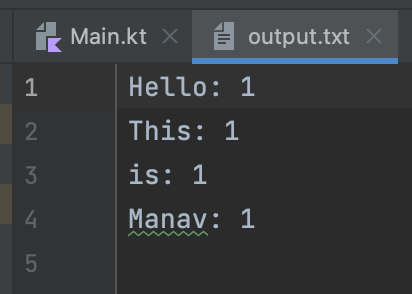
}

**}**

*println*("Word frequencies saved to output.txt.")

}

* We define the inputFile and outputFile variables representing the input and output files, respectively. Make sure to replace "input.txt" and "output.txt" with the actual file paths.
* We create a mutable map wordCountMap to store the word frequencies. The keys of the map are words, and the values are the corresponding counts.
* We read the input file line by line using forEachLine and split each line into individual words using a regular expression (\\s+ matches one or more whitespace characters). We trim each word to remove leading and trailing whitespaces.
* For each word, we update its count in the wordCountMap using getOrDefault to get the current count (defaulting to 0) and incrementing it by 1.
* We create a PrintWriter for the output file using the printWriter() function of outputFile. Inside the use block, we iterate over the wordCountMap and write each word with its count to the output file.
* Finally, we print a message indicating that the word frequencies have been saved to the output file.

# A program that demonstrates the use of coroutines to fetch data from two different web APIs concurrently.

import java.net.URL

fun fetchDataFromAPI(apiURL: String): String{ val url = URL(apiURL)

return url.*readText*()

}

fun main(){

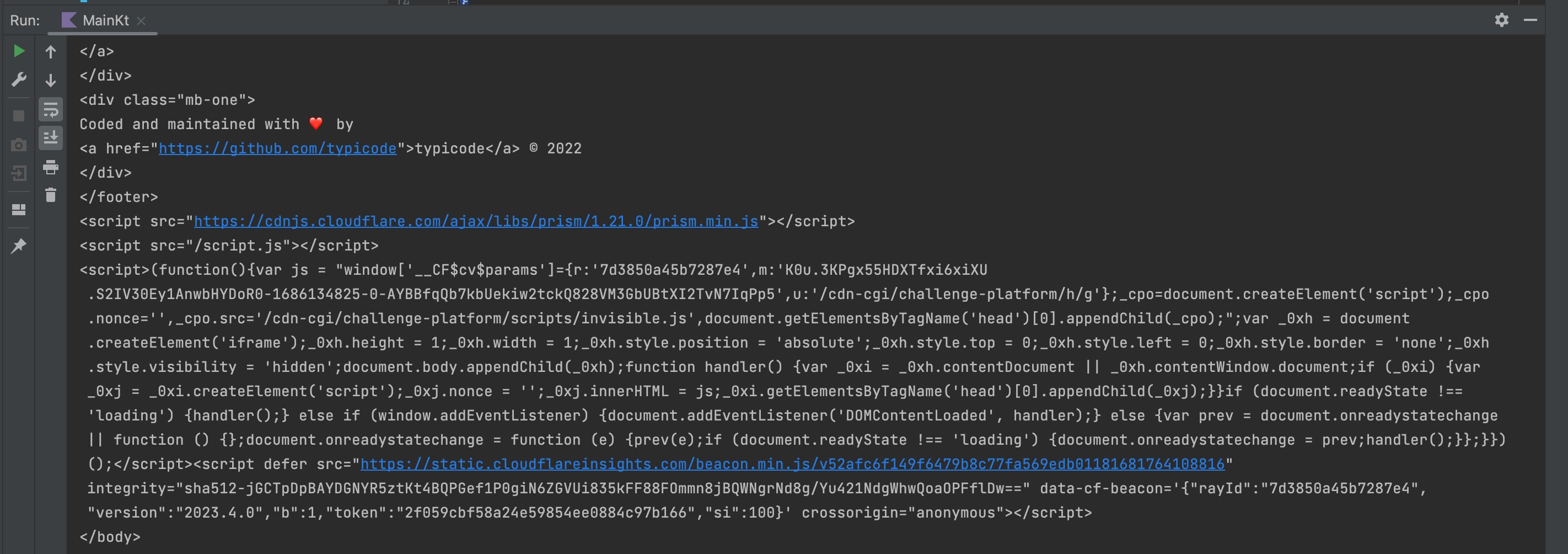
val apiUrl1 = "https://restcountries.com/v2/all" val data1 = *fetchDataFromAPI*(apiUrl1)

val apiUrl2 = "https://jsonplaceholder.typicode.com" val data2 = *fetchDataFromAPI*(apiUrl2)

*println*("API 1 response: $data1")

*println*("API 2 response: $data2")

}



# A program to find the GCD of two given numbers using the Euclidean algorithm.

fun findGCD(a: Int, b: Int): Int { var num1 = a

var num2 = b

while (num2 != 0) {

val remainder = num1 % num2 num1 = num2

num2 = remainder

}

return num1

}

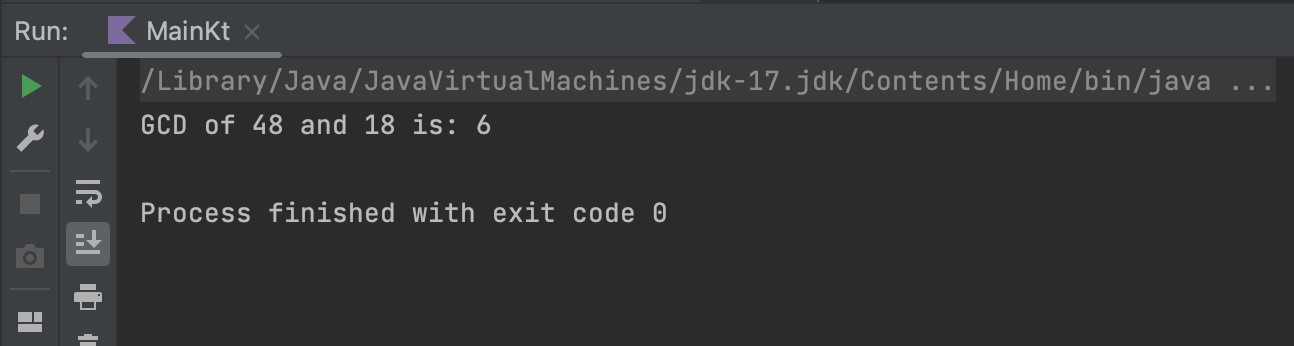
fun main() {

val num1 = 48 val num2 = 18

val gcd = *findGCD*(num1, num2)

*println*("GCD of $num1 and $num2 is: $gcd")

}



* The while loop continues until num2 becomes zero. In each iteration, we calculate the remainder of num1 divided by num2 using the modulus operator % and assign it to remainder. Then we update num1 with the value of num2 and num2 with the value of remainder. This process repeats until num2 becomes zero.
* Finally, we return the value of num1, which represents the GCD of the two input numbers.

# An extension function for the String class that returns the number of vowels in the string.

fun String.countVowels(): Int {

val vowels = *setOf*('a', 'e', 'i', 'o', 'u') var count = 0

for (char in this.*toLowerCase*()) { if (char in vowels) {

count++

}

}

return count

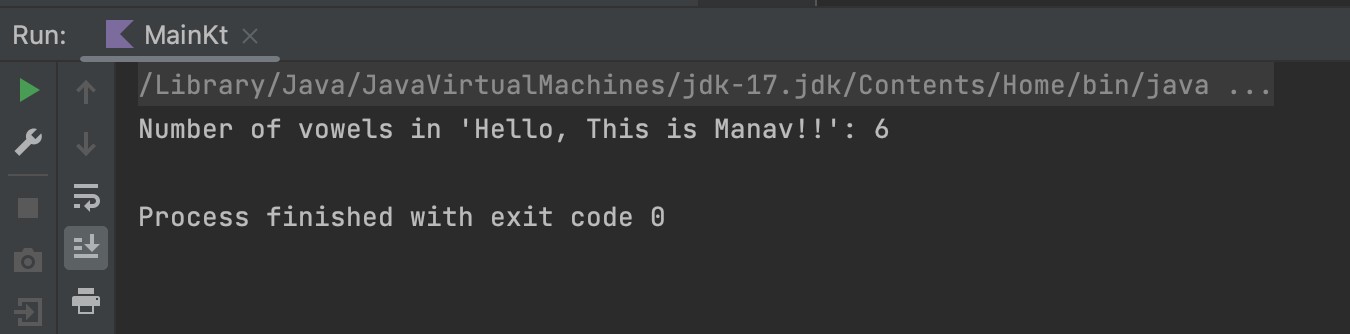
}

fun main() {

val text = "Hello, This is Manav!!" val vowelCount = text.*countVowels*()

*println*("Number of vowels in '$text': $vowelCount")

}



# A program to generate a list of the first n Fibonacci numbers, where n is a user input.

fun generateFibonacciNumbers(n: Int): List<Int> { val fibonacciNumbers = *mutableListOf*<Int>()

if (n >= 1) { fibonacciNumbers.add(0)

}

if (n >= 2) { fibonacciNumbers.add(1)

}

var a = 0 var b = 1

for (i in 3..n) {

val sum = a + b fibonacciNumbers.add(sum) a = b

b = sum

}

return fibonacciNumbers

}

fun main() {

*print*("Enter the value of n: ") val n = *readLine*()?.*toIntOrNull*()

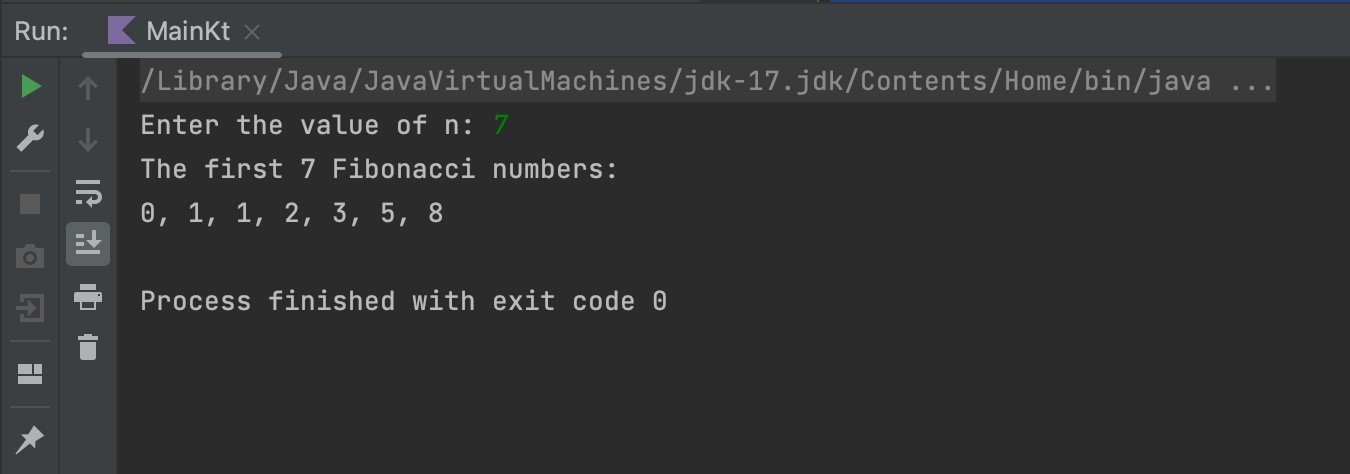
if (n != null && n > 0) {

val fibonacciNumbers = *generateFibonacciNumbers*(n) *println*("The first $n Fibonacci numbers:") *println*(fibonacciNumbers.*joinToString*(", "))

} else {

*println*("Invalid input. Please enter a positive integer.")

}

}

# A function that takes two integer arrays as arguments and returns a new array that contains the intersection of the two input arrays.

fun findArrayIntersection(arr1: IntArray, arr2: IntArray): IntArray { val set1 = arr1.*toSet*()

val set2 = arr2.*toSet*()

val intersection = set1.*intersect*(set2) return intersection.*toIntArray*()

}

fun main() {

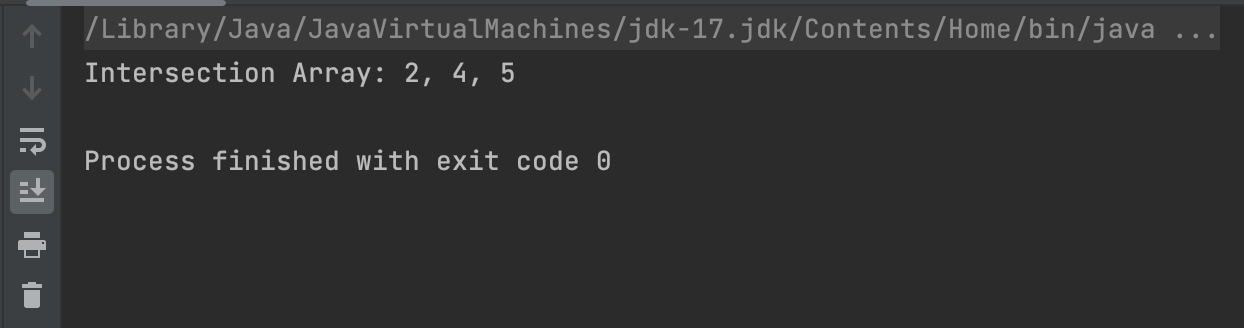
val array1 = *intArrayOf*(1, 2, 3, 4, 5)

val array2 = *intArrayOf*(4, 5, 6, 7, 8, 2)

val intersectionArray = *findArrayIntersection*(array1, array2)

*println*("Intersection Array: ${intersectionArray.*joinToString*(", ")}")

}



# A program that creates a Person class with properties: firstName, lastName, and age.

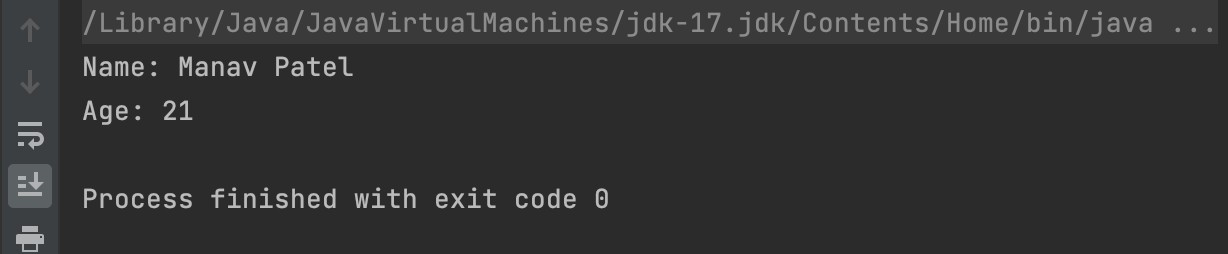
class Person(val firstName: String, val lastName: String, val age: Int) fun main() {

val person = Person("Manav", "Patel", 21)

*println*("Name: ${person.firstName} ${person.lastName}")

*println*("Age: ${person.age}")

}



# A program that creates a sealed class named Result and two subclasses Success and Failure.

sealed class Result

data class Success(val message: String) : Result() data class Failure(val errorMessage: String) : Result()

fun processResult(result: Result) { when (result) {

is Success -> *println*("Success: ${result.message}") is Failure -> *println*("Failure: ${result.errorMessage}")

}

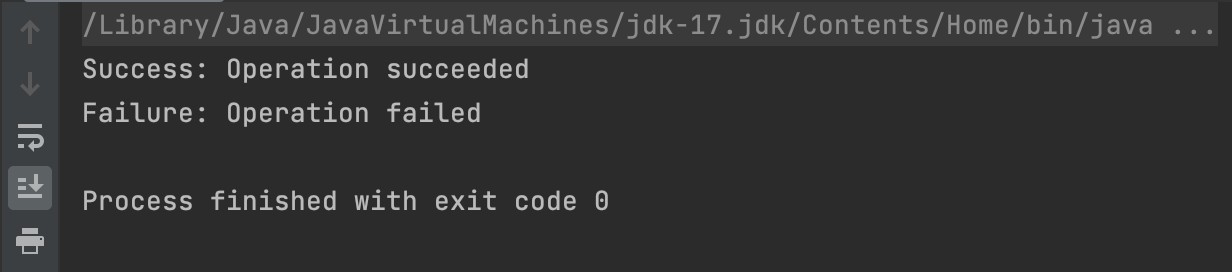
}

fun main() {

val successResult = Success("Operation succeeded") val failureResult = Failure("Operation failed")

*processResult*(successResult) *processResult*(failureResult)

}



# A program that uses anonymous functions and higher-order functions to calculate the sum of all even numbers in a list of integers.

fun main() {

val numbers = *listOf*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

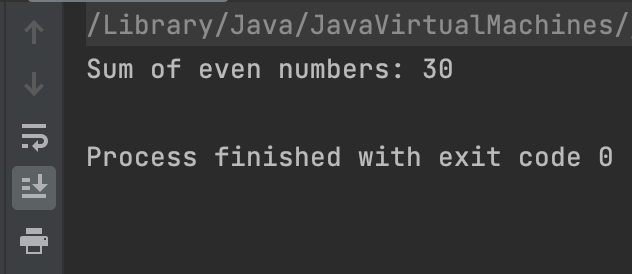
val evenSum = numbers.*filter*(fun(num): Boolean { return num % 2 == 0

}).*sum*()

*println*("Sum of even numbers: $evenSum")

}

* In this program, we have a list of integers named numbers containing some values.
* We use the filter higher-order function on the numbers list. Inside the filter function, we define an anonymous function using the fun keyword. This anonymous function takes an integer num as input and returns a boolean value indicating whether num is even (i.e., num % 2 == 0).
* The filter function filters out all the elements from the numbers list for which the anonymous function returns true, i.e., the even numbers.
* Finally, we use the sum function on the filtered list to calculate the sum of all the even numbers.
* The result, evenSum, is then printed to the console.



# A program that creates an abstract class Vehicle with two subclasses: Car and Motorcycle.

abstract class Vehicle(val brand: String, val model: String) { abstract fun start()

abstract fun stop()

}

class Car(brand: String, model: String) : Vehicle(brand, model) { override fun start() {

*println*("Starting the $brand $model car.")

}

override fun stop() {

*println*("Stopping the $brand $model car.")

}

}

class Motorcycle(brand: String, model: String) : Vehicle(brand, model) { override fun start() {

*println*("Starting the $brand $model motorcycle.")

}

override fun stop() {

*println*("Stopping the $brand $model motorcycle.")

}

}

fun main() {

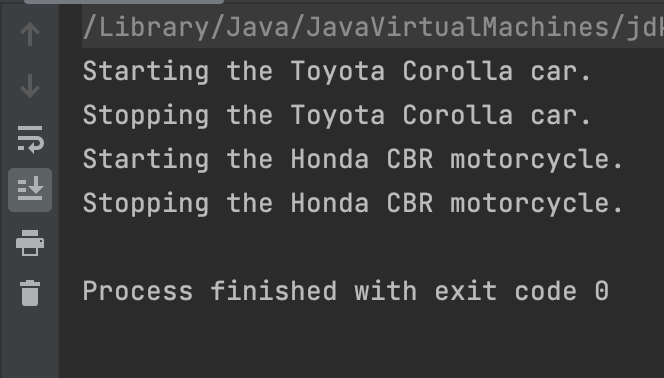
val car = Car("Toyota", "Corolla") car.start()

car.stop()

val motorcycle = Motorcycle("Honda", "CBR") motorcycle.start()

motorcycle.stop()

}



1. A program to read a CSV file, filter the records based on a given condition, and save the filtered records in a new CSV file.
2. A program that demonstrates the use of Kotlin Flow to emit a sequence of integers and perform a transformation on each emitted value.