Kotlin Syntax

Produced by: Dr. Siobhán Drohan (sdrohan@wit.ie)

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

Kotlin Syntax

Sources:

- http://kotlinlang.org/docs/reference/basic-syntax.html
- http://petersommerhoff.com/dev/kotlin/kotlin-for-java-devs/
- https://www.programiz.com/kotlin-programming
- https://medium.com/@napperley/kotlin-tutorial-5-basic-collections-3f114996692b

Agenda from last Kotlin Lecture

- *Basic Types*
- *Local Variables (val & var)*
- *Functions*
- *Control Flow (if, when, for, while)*
- *Strings & String Templates*
- *Ranges (and the **in** operator)*
- *Type Checks & Casts*
- *Null Safety*
- *Comments*

Agenda for this Kotlin Lecture

- Writing Classes (properties and fields)
- Data Classes
- Collections: Arrays and Collections
- Collections: *in* operator and lambdas
- Arguments (default and named)

Kotlin
by JetBrains

Writing Classes

Properties and Fields

Writing Classes – properties

In Kotlin, classes cannot have fields; they have properties.

var properties are mutable.
val properties cannot be changed.

Writing Classes – constructors

A class in Kotlin can have a **primary constructor** and one or more **secondary constructors**.

The **primary constructor** is part of the class header and it goes after the class name:

```
class Person constructor(firstName: String) {
}
```

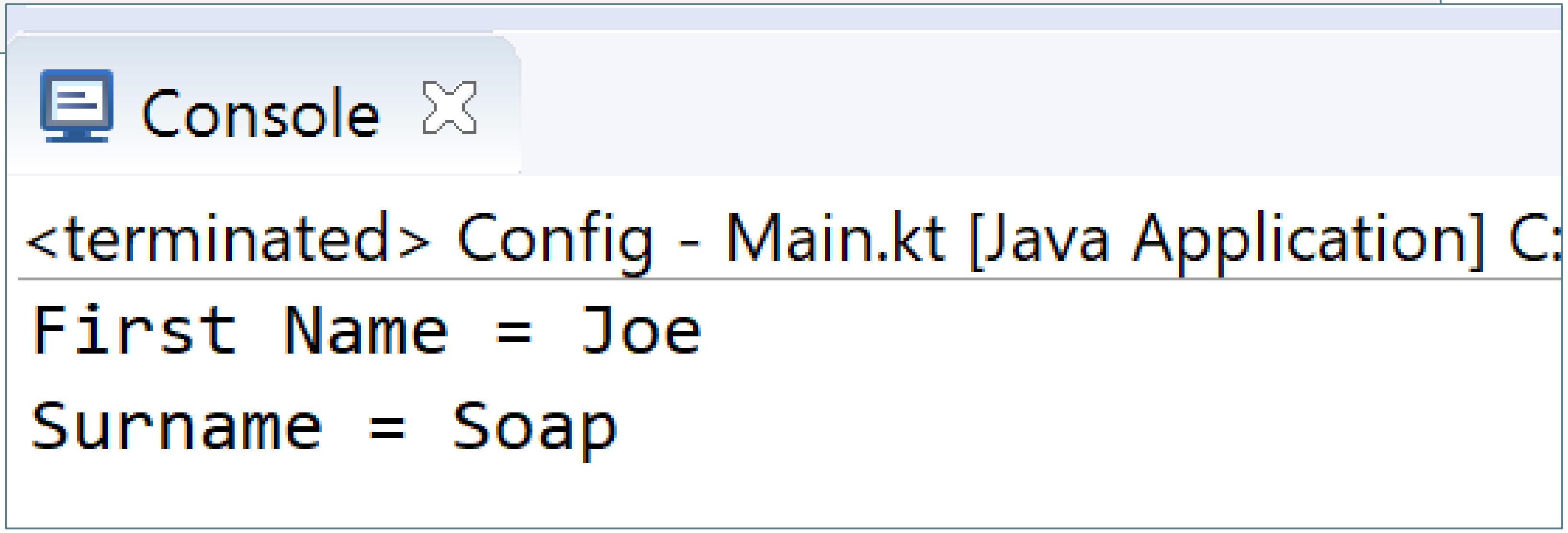
```
class Person(firstName: String, lastName: String) {
}
```

```
class Person(val firstName: String, val lastName: String) {
}
```

Writing Classes – primary constructors

```
class Person(val firstName: String, val lastName: String) {

}
```

```
fun main(args: Array<String>) {

    val person = Person("Joe", "Soap")

    println("First Name = ${person.firstName}")
    println("Surname = ${person.lastName}")
}
```



```
<terminated> Config - Main.kt [Java Application] C:
First Name = Joe
Surname = Soap
```

Writing Classes – primary constructors

```
class Person( _firstName: String = "UNKNOWN FIRSTNAME",
              _lastName: String = "UNKNOWN LASTNAME") {

    val firstName = _firstName
    val lastName  = _lastName

    // initializer block
    init {
        println("First Name = $firstName")
        println("Last Name = $lastName\n")
    }
}
```

The **primary constructor** cannot contain any code; initialisation code is placed in the **init** block.

The use of _ prefixing constructor variables is standard.

Writing Classes – primary constructors

```
class Person( _firstName: String = "UNKNOWN FIRSTNAME",
              _lastName: String = "UNKNOWN LASTNAME") {

    val firstName = _firstName
    val lastName  = _lastName

    // initializer block
    init {
        println("First Name = $firstName")
        println("Last Name = $lastName\n")
    }
}
```

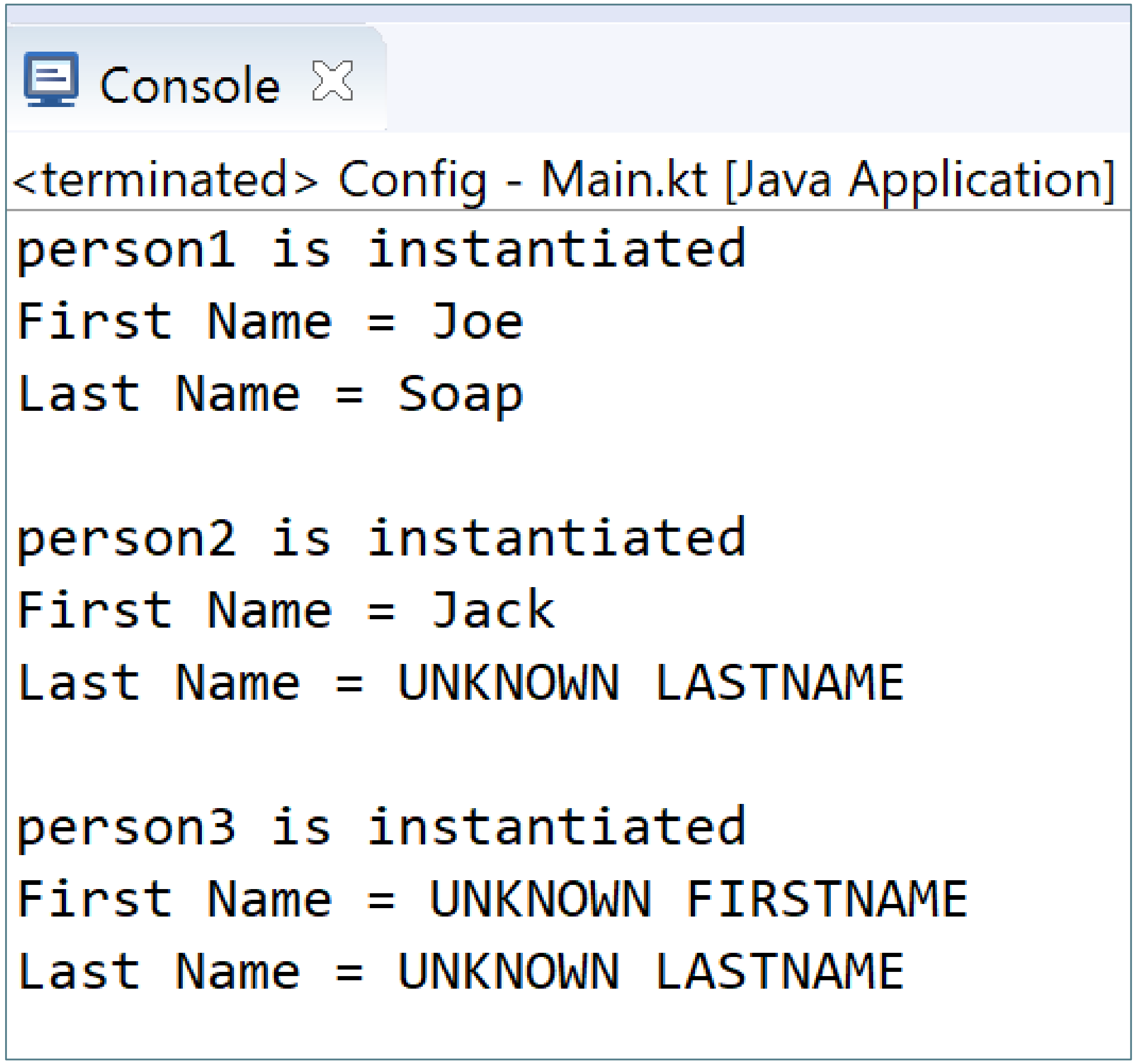
```
fun main(args: Array<String>) {

    println("person1 is instantiated")
    val person1 = Person("Joe", "Soap")

    println("person2 is instantiated")
    val person2 = Person("Jack")

    println("person3 is instantiated")
    val person3 = Person()
}
```

Writing Classes – primary constructors

```
class Person( _firstName: String = "UNKNOWN FIRSTNAME",
              _lastName: String = "UNKNOWN LASTNAME") {

    val firstName = _firstName
    val lastName  = _lastName

    // initializer block
    init {
        println("First Name = $firstName")
        println("Last Name = $lastName\n")
    }
}
```

```
fun main(args: Array<String>) {

    println("person1 is instantiated")
    val person1 = Person("Joe", "Soap")

    println("person2 is instantiated")
    val person2 = Person("Jack")

    println("person3 is instantiated")
    val person3 = Person()
}
```

Note: varied parameters allowed in primary constructor as values are defaulted (i.e. optional parameters)



```
<terminated> Config - Main.kt [Java Application]
person1 is instantiated
First Name = Joe
Last Name = Soap

person2 is instantiated
First Name = Jack
Last Name = UNKNOWN LASTNAME

person3 is instantiated
First Name = UNKNOWN FIRSTNAME
Last Name = UNKNOWN LASTNAME
```

Writing Classes – secondary constructors

The **secondary constructor** is prefixed with the keyword **constructor**. They are not very common in Kotlin.

More info here:
http://kotlinlang.org/docs/reference/classes.html

```
class Person {

    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

Writing Classes – getters and setters

In Kotlin, getters (val and var) and setters (var) are optional and are auto-generated if you do not create them in your program.

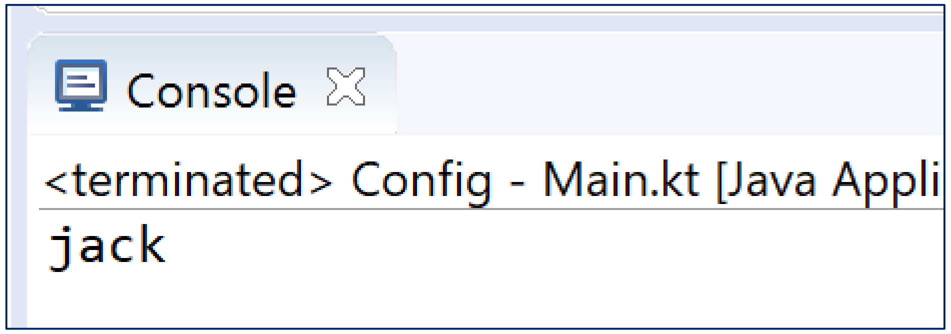
```kotlin
class Person {
    var name: String = "defaultValue"
}
```

Is equivalent to

```kotlin
class Person {
    var name: String = "defaultValue"

    // getter
    get() = field

    // setter
    set(value) {
        field = value
    }
}
```

Writing Classes – getters and setters

```
fun main(args: Array<String>) {

    val person = Person()
    person.name = "jack"
    print(person.name)

}
```

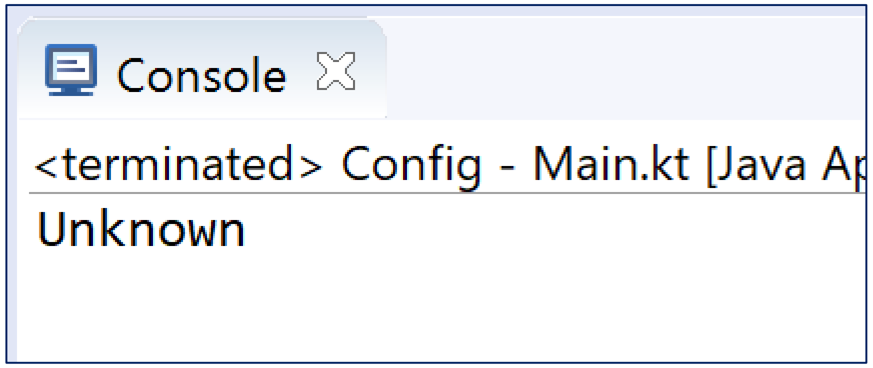


```
<terminated> Config - Main.kt [Java Appli
jack
```

```
class Person {
    var name: String = "defaultValue"

    // getter
    get() = field

    // setter
    set(value) {
        field = value
    }
}
```

Writing Classes – getters and setters

```
fun main(args: Array<String>) {

    val person = Person()
    person.name = ""
    print(person.name)

}
```



```
<terminated> Config - Main.kt [Java Ap
Unknown
```

When you want to add validation to your setter…

```
class Person {
    var name: String = "defaultValue"

    get() = field

    set(value) {
            field = if (value.equals(""))
                       "Unknown"
                    else
                       value
     }
}
```

Data Classes

Data Classes

We have created classes to solely to hold data (i.e. models).

We can use the **data** class prefix to simply create a data class.

The compiler automatically generates methods such as **equals(), hashCode(), toString(), copy()** from the primary constructor.

```
data class Person( var firstName: String,
                   var lastName: String)
{
}
```

Data Classes - Requirements

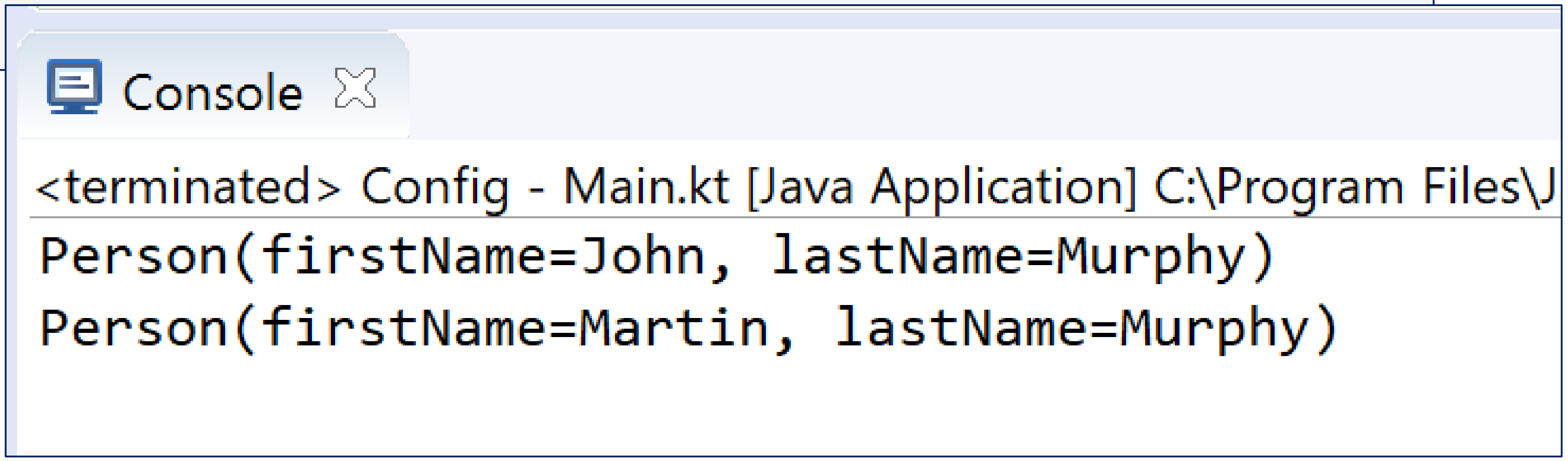
1. The primary constructor must have at least one parameter
2. The parameters of the primary constructor must be marked as either var or val
3. The class cannot be open, abstract, inner or sealed
4. The class may extend other classes or implement interfaces

```
data class Person( var firstName: String,
                   var lastName: String)
{
}
```

Data Classes – copy and toString Example

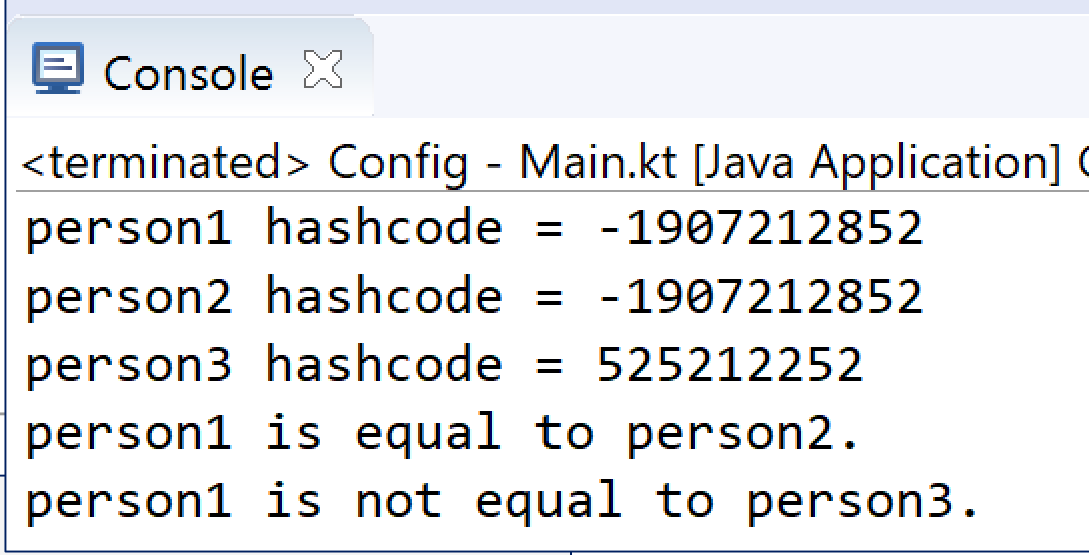
```
data class Person( var firstName: String,
                           var lastName: String){
}
```

```
fun main(args: Array<String>) {
    val person1 = Person("John", "Murphy")

    // using copy function to create an object
    val person2 = person1.copy(firstName="Martin")

    println(person1)
    println(person2.toString())
}
```



```
<terminated> Config - Main.kt [Java Application] C:\Program Files\J
Person(firstName=John, lastName=Murphy)
Person(firstName=Martin, lastName=Murphy)
```

Data Classes – copy, equals and hashCode Example



<terminated> Config - Main.kt [Java Application] C

```
person1 hashcode = -1907212852
person2 hashcode = -1907212852
person3 hashcode = 525212252
person1 is equal to person2.
person1 is not equal to person3.
```

```kotlin
fun main(args: Array<String>) {
    val person1 = Person("John", "Murphy")
    val person2 = person1.copy()
    val person3 = person1.copy(firstName = "Martin")

    println("person1 hashcode = ${person1.hashCode()}")
    println("person2 hashcode = ${person2.hashCode()}")
    println("person3 hashcode = ${person3.hashCode()}")

    if (person1.equals(person2))
        println("person1 is equal to person2.")
    else
        println("person1 is not equal to person2.")

    if (person1.equals(person3))
        println("person1 is equal to person3.")
    else
        println("person1 is not equal to person3.")
}
```

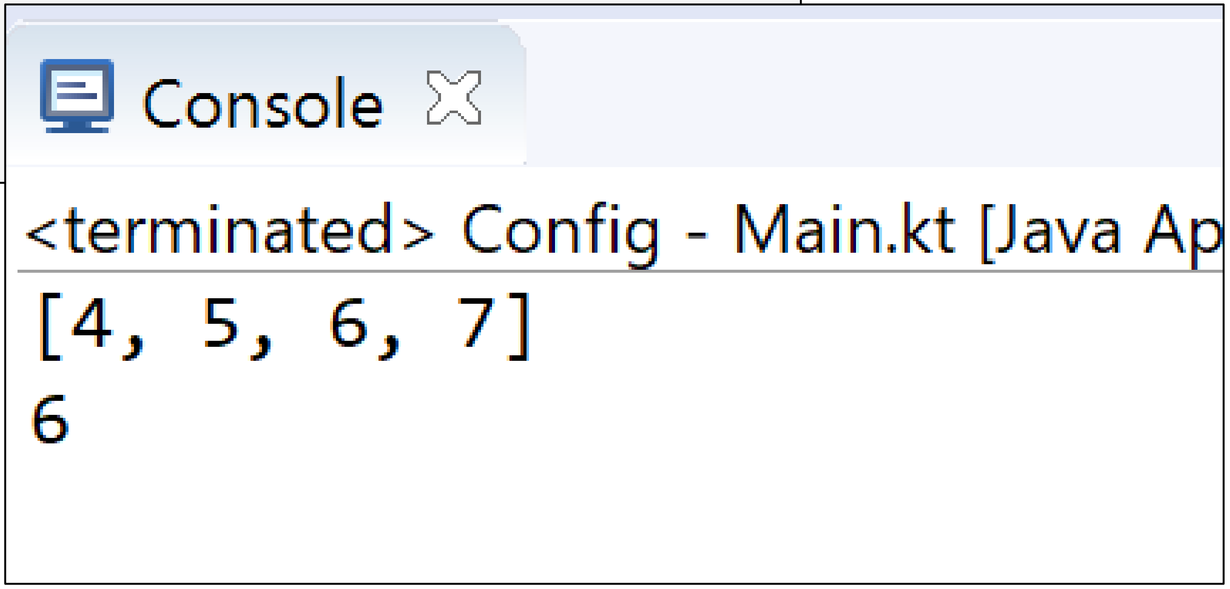
Collections

Arrays and Collections

Arrays (using arrayOf)

Arrays in Kotlin can be created using **arrayOf()** or the **Array()** constructor.

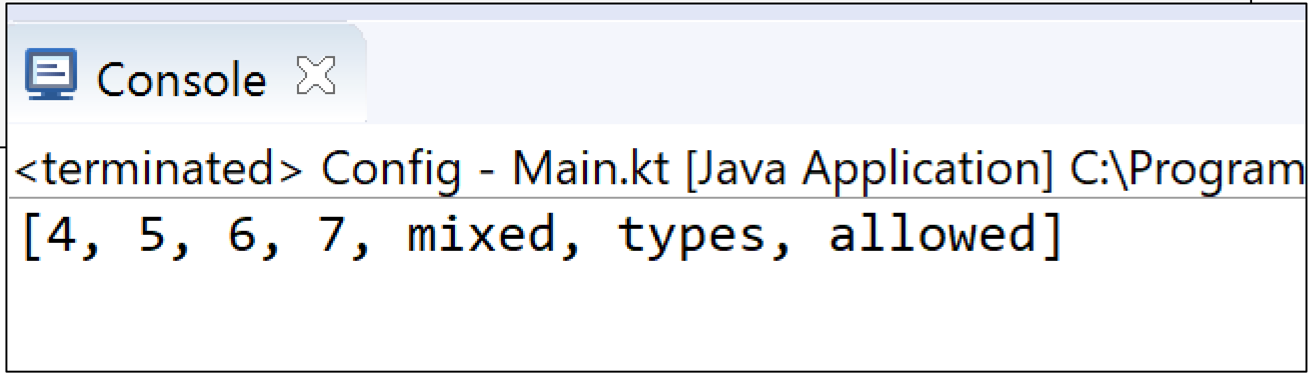
```
fun main(args: Array<String>) {

  val myArray = arrayOf(4, 5, 6, 7)
  println(myArray.asList())
  print(myArray[2])

}
```



```
<terminated> Config - Main.kt [Java Ap
[4, 5, 6, 7]
6
```

Arrays (using arrayOf)

```
fun main(args: Array<String>) {

   val myArray = arrayOf(4, 5, 6, 7, "mixed", “types", "allowed")
   print(myArray.asList())
}
```

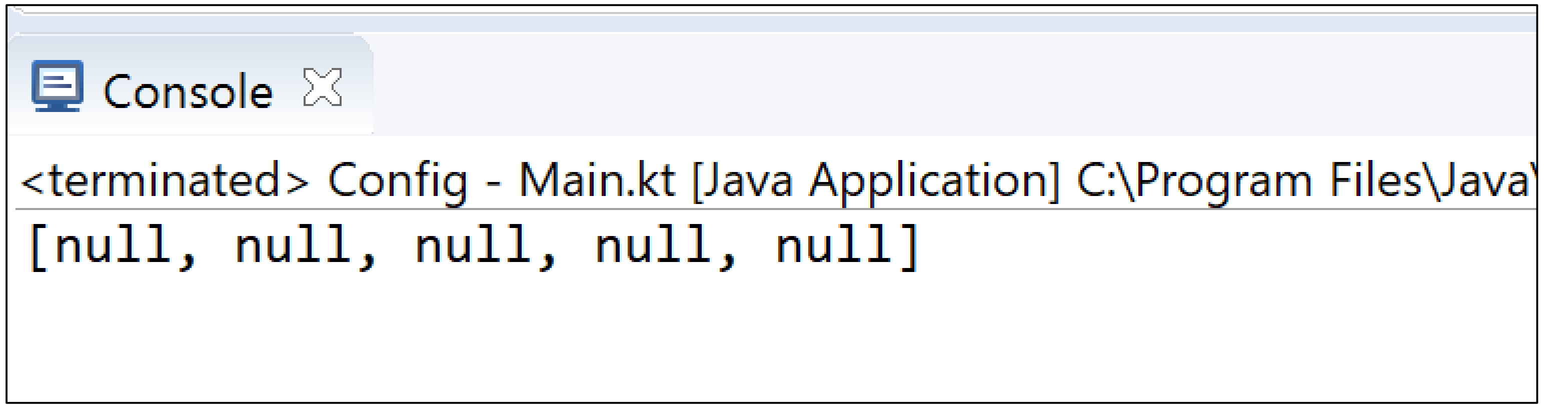


```
<terminated> Config - Main.kt [Java Application] C:\Program
[4, 5, 6, 7, mixed, types, allowed]
```

```
fun main(args: Array<String>) {

  val intArray1     = intArrayOf(4, 5, 6, 7)
  val intArray2     = arrayOf<Int>(4, 5, 6, 7)
  val charArray     = charArrayOf('a', 'b', 'c', 'd’)
  val booleanArray = booleanArrayOf(true, false, true)

  val mixedArray1 = intArrayOf(4, 5, 6, 7, "will","not","compile")
  val mixedArray2 = arrayOf<Int>(4, 5, 6, 7,"will","not","compile")

}
```

Arrays (using arrayOfNulls)

```kotlin
fun main(args: Array<String>) {

   val nullArray = arrayOfNulls<Int>(5);
   println (nullArray.asList())

}
```



```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\
[null, null, null, null, null]
```

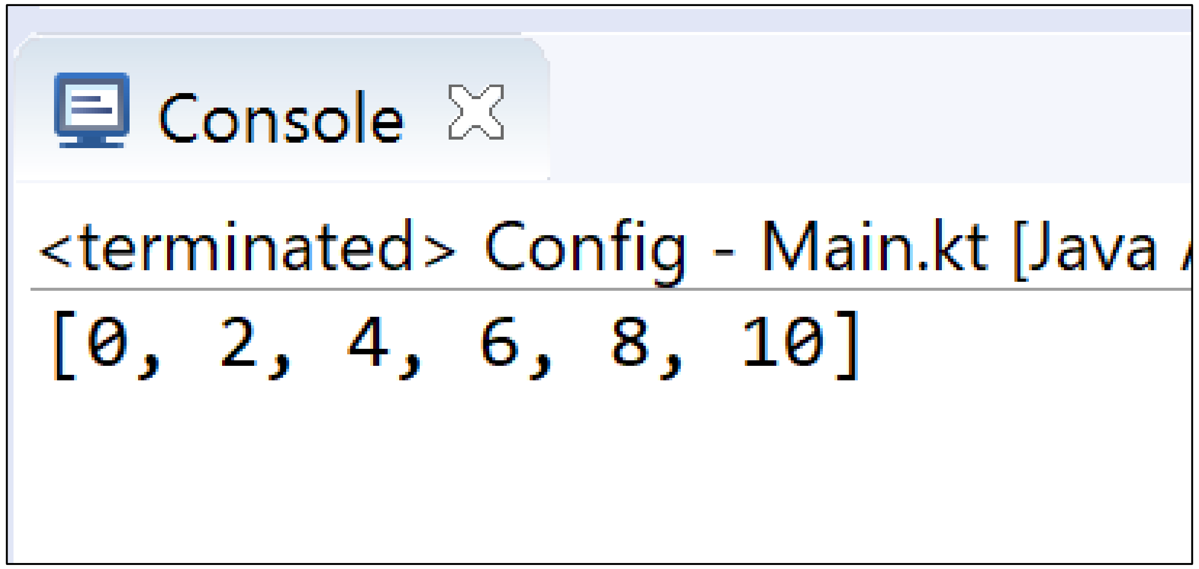
Arrays (using constructor)

The **Array()** constructor requires a size and a lambda function.

```
fun main(args: Array<String>) {

   val intArray = Array(6, { i -> i * 2 })
   print (intArray.asList())

}
```

Index of the array element

Value to be inserted into the index



```
<terminated> Config - Main.kt [Java
[0, 2, 4, 6, 8, 10]
```

Collections

Unlike many languages, Kotlin distinguishes between **mutable** and **immutable** collections (lists, sets, maps, etc).

Precise control over exactly when collections can be edited is useful for eliminating bugs, and for designing good APIs.

Collections – mutable vs immutable

The Kotlin List<out T> type is an interface that provides read-only operations like size, get and so on.

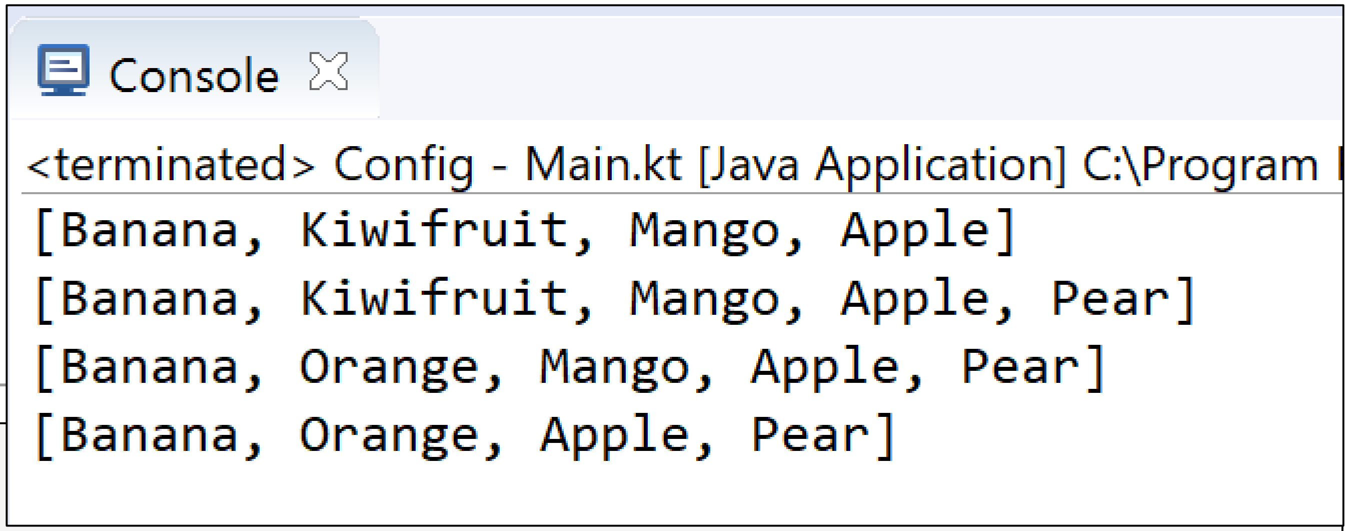
Like in Java, it inherits from Collection<T> and that in turn inherits from Iterable<T>.

Methods that change the list are added by the MutableList<T> interface.

This pattern holds also for Set<out T>/MutableSet<T> and Map<K, out V>/MutableMap<K, V>.

Collections – mutable List

```kotlin
fun main(args: Array<String>) {

  // Create a mutable list (MutableList).
  val fruit = mutableListOf("Banana", "Kiwifruit", "Mango", "Apple")
  println(fruit)

  // Add a element to the list.
  fruit.add("Pear")
  println(fruit)

  // Change an element in the list.
  fruit[1] = "Orange"
  println(fruit)

  // Remove a existing element from the list.
  fruit.removeAt(2)
  println(fruit)

}
```



<terminated> Config - Main.kt [Java Application] C:\Program

```
[Banana, Kiwifruit, Mango, Apple]
[Banana, Kiwifruit, Mango, Apple, Pear]
[Banana, Orange, Mango, Apple, Pear]
[Banana, Orange, Apple, Pear]
```

Collections – immutable List – example 1

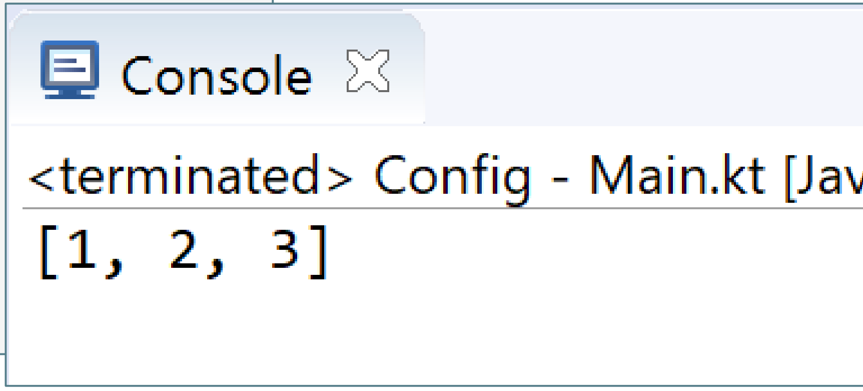
```
fun main(args: Array<String>) {

   val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
   val readOnlyView: List<Int> = numbers

   println(numbers)        // prints "[1, 2, 3]"
   numbers.add(4)

   println(readOnlyView)   // prints "[1, 2, 3, 4]"
   readOnlyView.clear()    // -> does not compile

}
```

Collections – immutable List – example 2

```
class Person(  _firstName: String = "UNKNOWN",
               _lastName: String = "UNKNOWN") {

   private val _items = mutableListOf<String>("1", "2", "3")
   val items: List<String> get() = _items.toList()
}
```

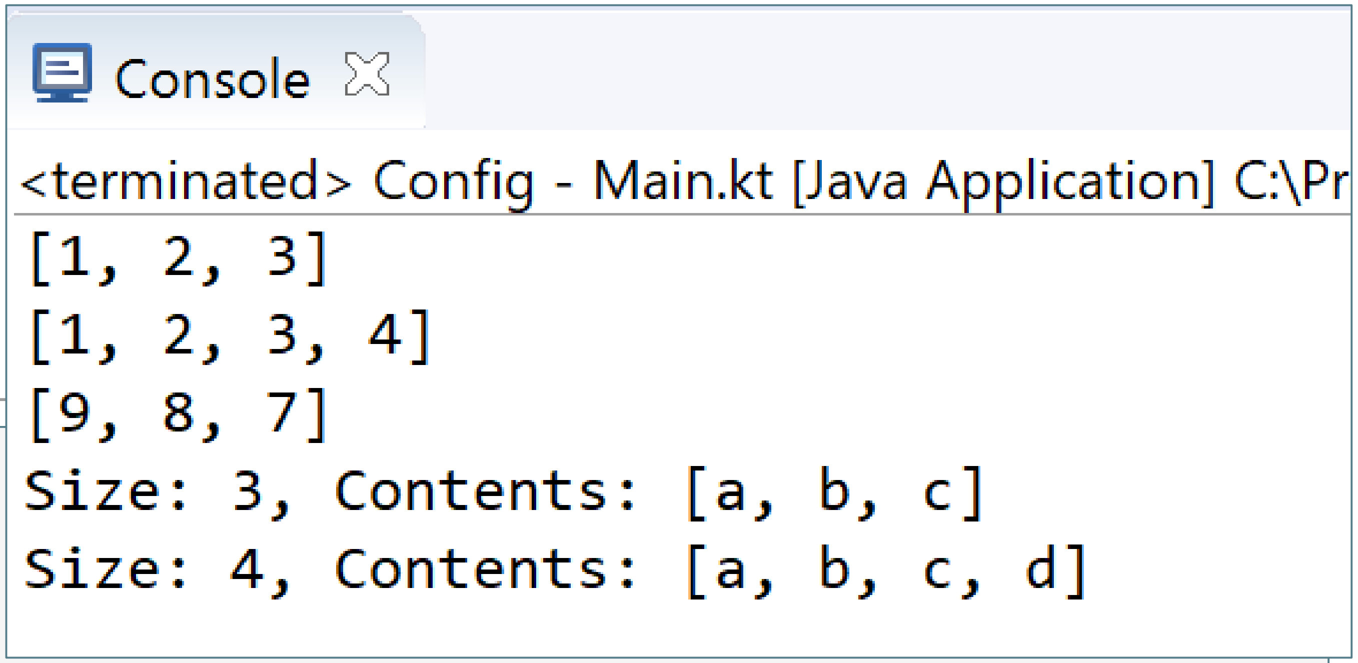
items returns a snapshot of a collection at a particular point in time (that's guaranteed to not change). **toList()** just duplicates the items.

```
fun main(args: Array<String>) {

   val person = Person()
   println(person.items)

   //person.items.clear()  //doesn't compile

}
```



```
<terminated> Config - Main.kt [Jav
[1, 2, 3]
```

Collections –
Set and hashSet

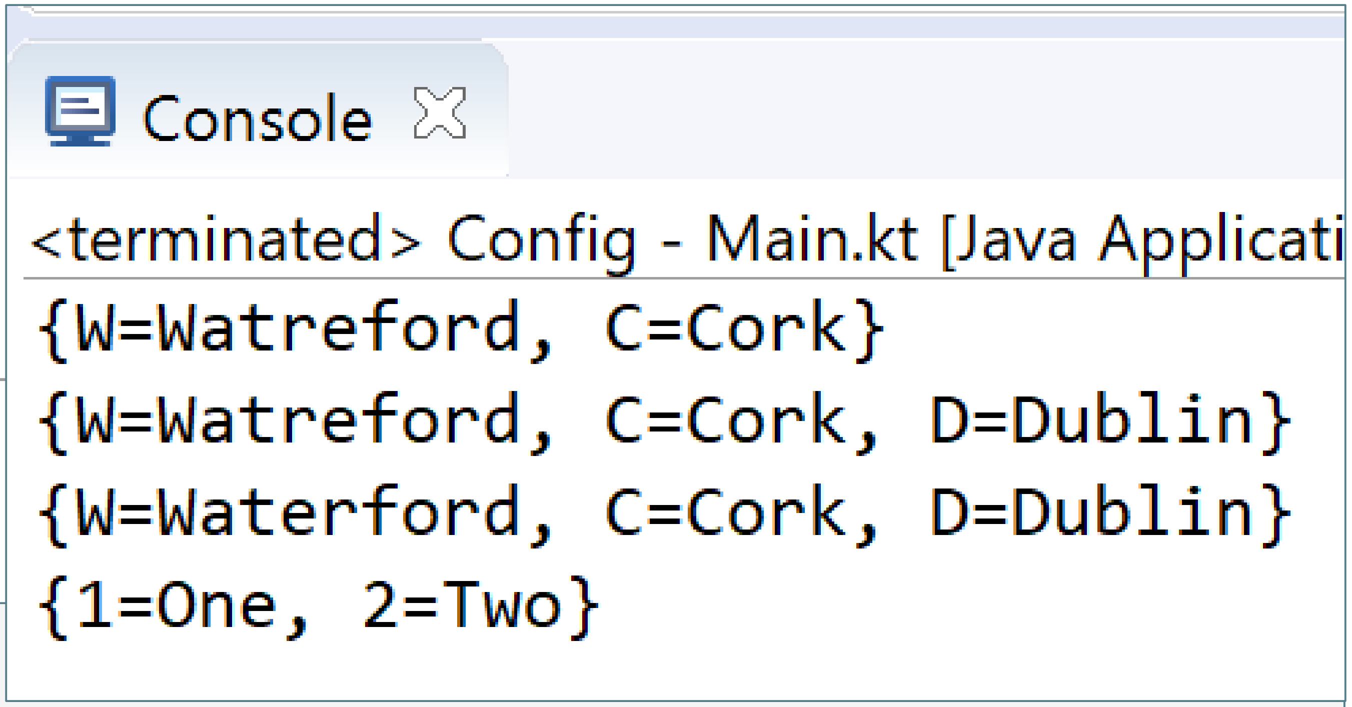


```
<terminated> Config - Main.kt [Java Application] C:\Pr
[1, 2, 3]
[1, 2, 3, 4]
[9, 8, 7]
Size: 3, Contents: [a, b, c]
Size: 4, Contents: [a, b, c, d]
```

```kotlin
fun main(args: Array<String>) {

    // mutatble set
    val mutableSet : MutableSet<Int> = mutableSetOf(1,2,3)
    println(mutableSet)
    mutableSet.add(4)
    println(mutableSet)

    // immutatble set
    val immutableSet : Set<Int> = setOf(9,8,7)
    println(immutableSet)
    //immutableSet.add(6)  //won't compile

    //note: ignores duplicate items
    val strings = hashSetOf("a", "b", "c", "c")
    println("Size: ${strings.size}, Contents: " + strings)
    strings.add("d")
    println("Size: ${strings.size}, Contents: " + strings)

}
```

Collections –
Map and hashMap

```
fun main(args: Array<String>) {

   // mutatble map
   val mutableMap = mutableMapOf("W" to "Watreford", "C" to "Cork")
   println(mutableMap)
   mutableMap.put("D", "Dublin")
   println(mutableMap)
   mutableMap["W"] = "Waterford"
   println(mutableMap)

   // immutatble map
   val immutableMap : Map<Int, String> = mapOf(1 to "One", 2 to "Two")
   println(immutableMap)
   //immutableMap.put(3, "Three")   //won't compile

}
```

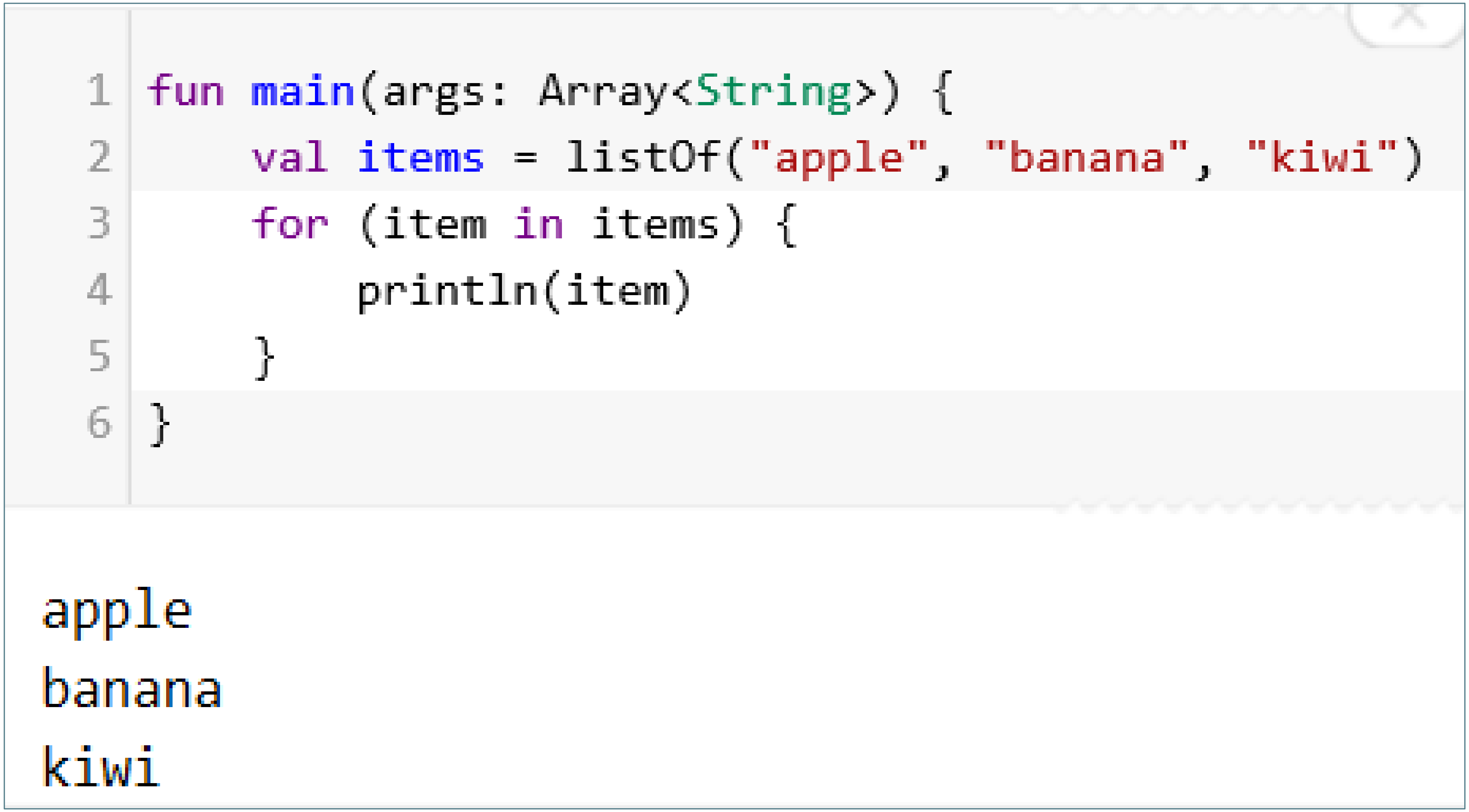


```
<terminated> Config - Main.kt [Java Applicati
{W=Watreford, C=Cork}
{W=Watreford, C=Cork, D=Dublin}
{W=Waterford, C=Cork, D=Dublin}
{1=One, 2=Two}
```

Collections

The in operator and using lambdas

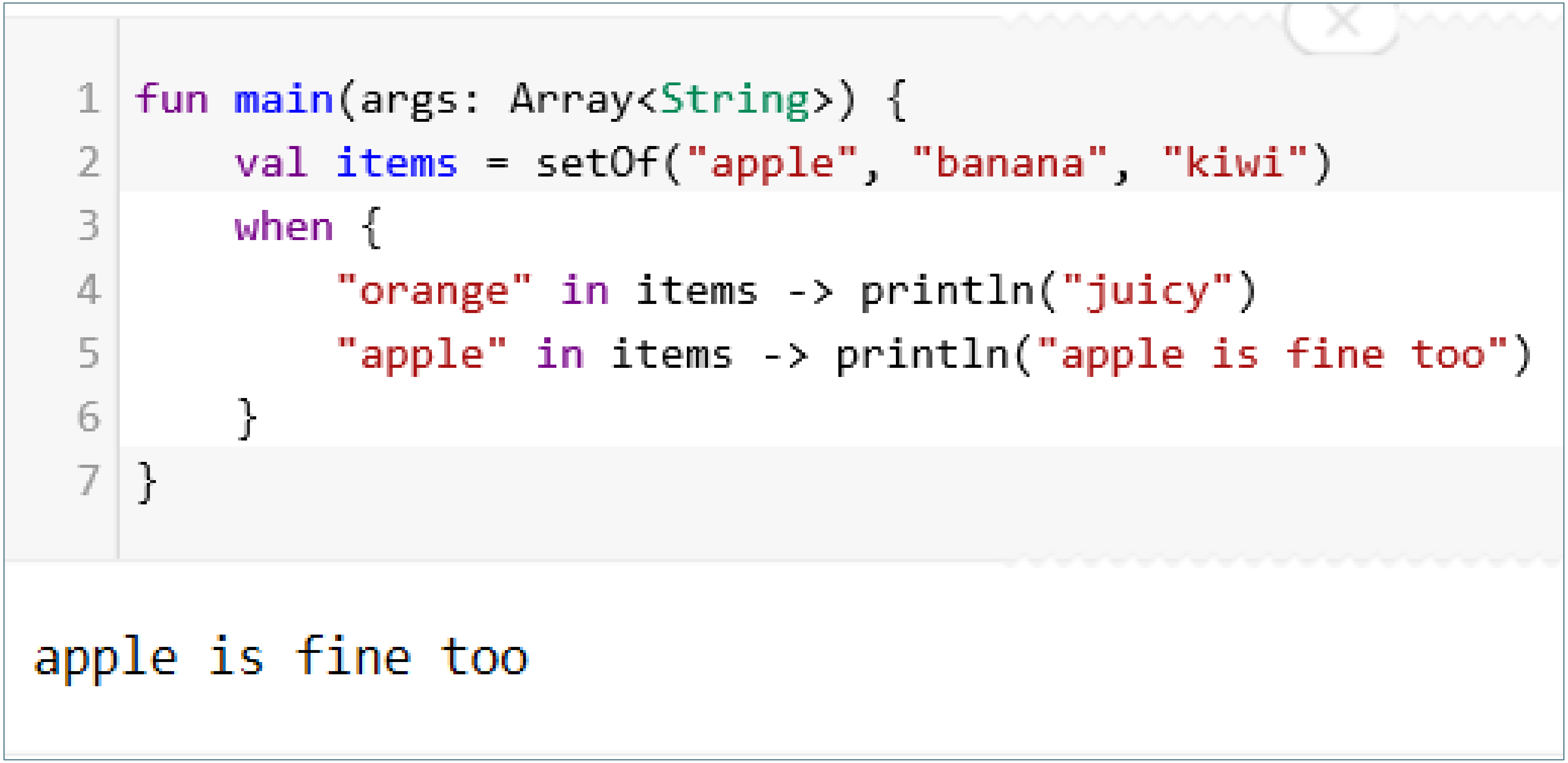
Collections – iterating using the **in** operator



```kotlin
fun main(args: Array<String>) {
    val items = listOf("apple", "banana", "kiwi")
    for (item in items) {
        println(item)
    }
}
```

```
apple
banana
kiwi
```

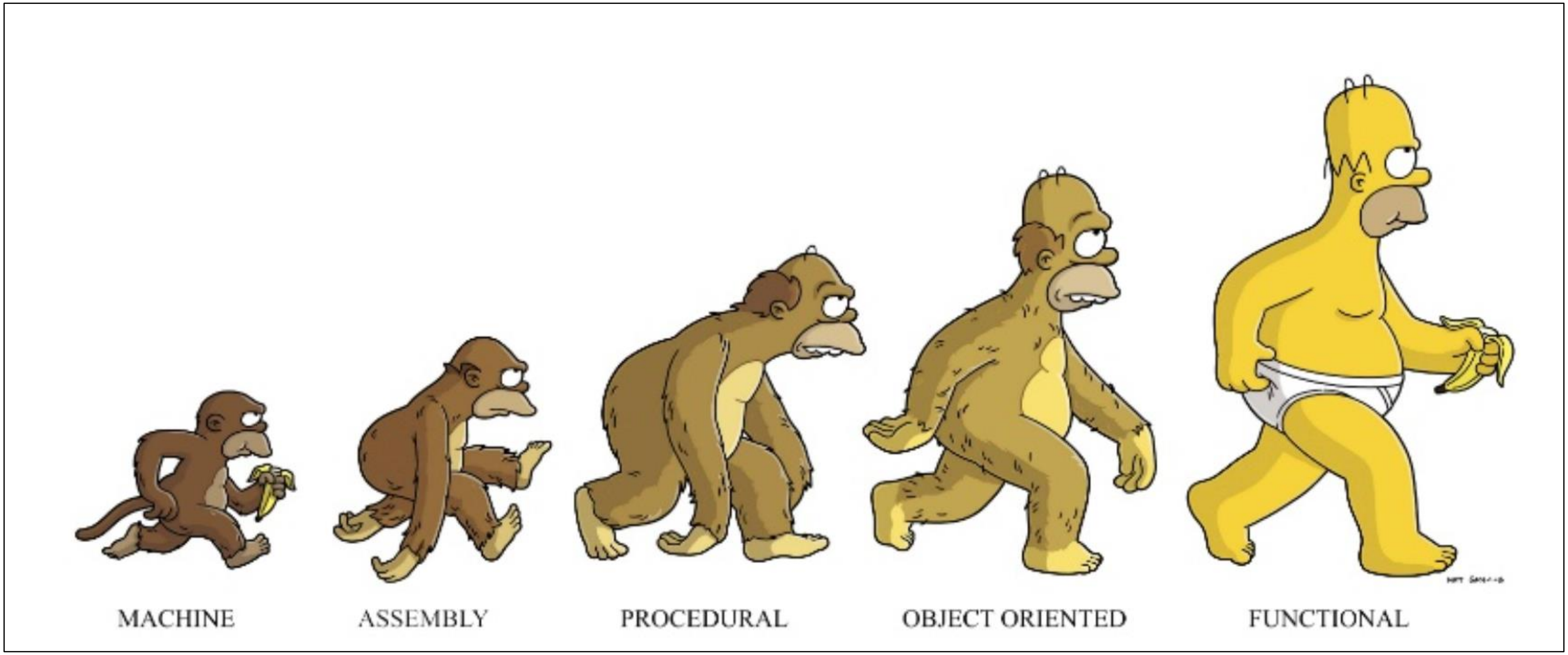
Collections – checking if collection contains an object



```kotlin
fun main(args: Array<String>) {
    val items = setOf("apple", "banana", "kiwi")
    when {
        "orange" in items -> println("juicy")
        "apple" in items -> println("apple is fine too")
    }
}
```

```
apple is fine too
```

Kotlin….functional programming is prevalent!



Collections – lambdas

- You can pass an anonymous function (a lambda) as a parameter of a function.

- A lambda expression is always surrounded by curly braces.

- Its parameters (if any) are declared before **->** (parameter types may be omitted),

- The body goes after **->** (when present).

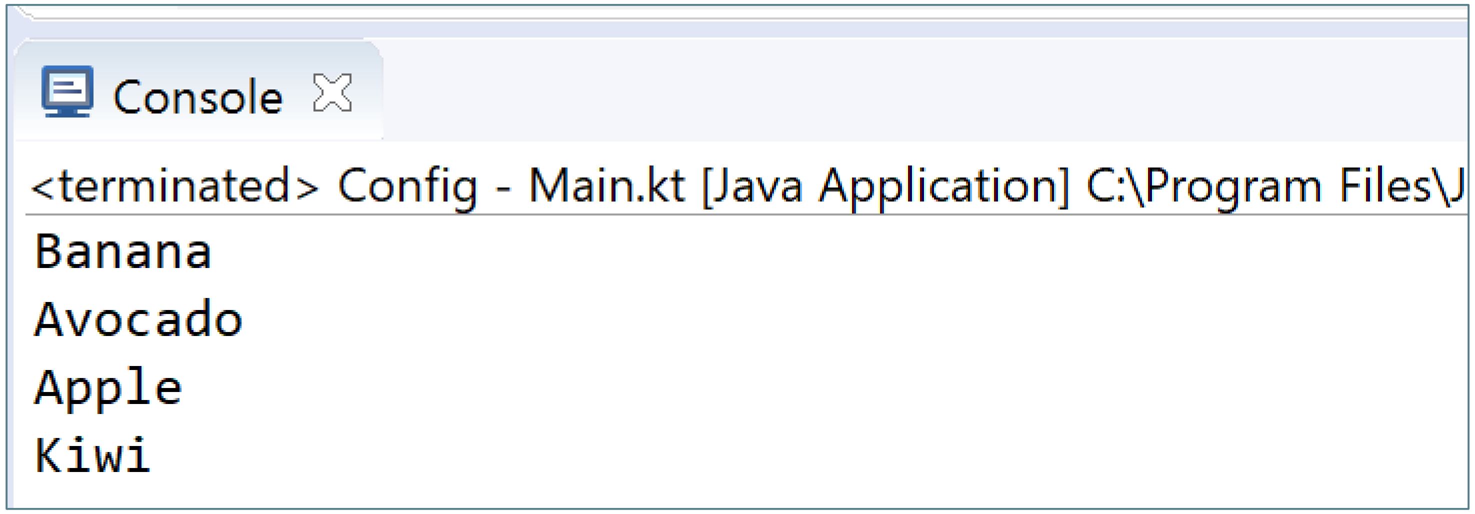
- An implicit variable called “it” is created and refers to the lambda expression’s only argument.

Collections – lambdas

Using lambda expressions to filter and map collections

```
fun main(args: Array<String>) {

  val fruits = listOf ("Banana", "Avocado", "Apple", "Kiwi")
  fruits.forEach  {it -> println(it)}
}
```

it -> is optional

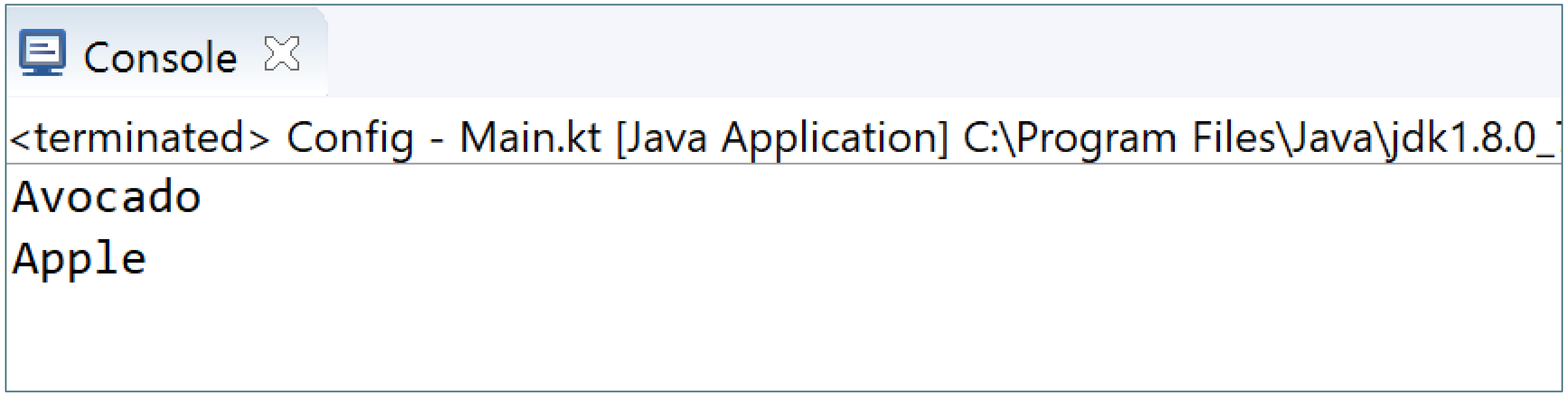


```
<terminated> Config - Main.kt [Java Application] C:\Program Files\J
Banana
Avocado
Apple
Kiwi
```

Collections – lambdas

Using lambda expressions to filter and map collections

```
fun main(args: Array<String>) {

  val fruits = listOf ("Banana", "Avocado", "Apple", "Kiwi")
  fruits.filter   {it.startsWith("A")
        .forEach  {println(it)}
}
```

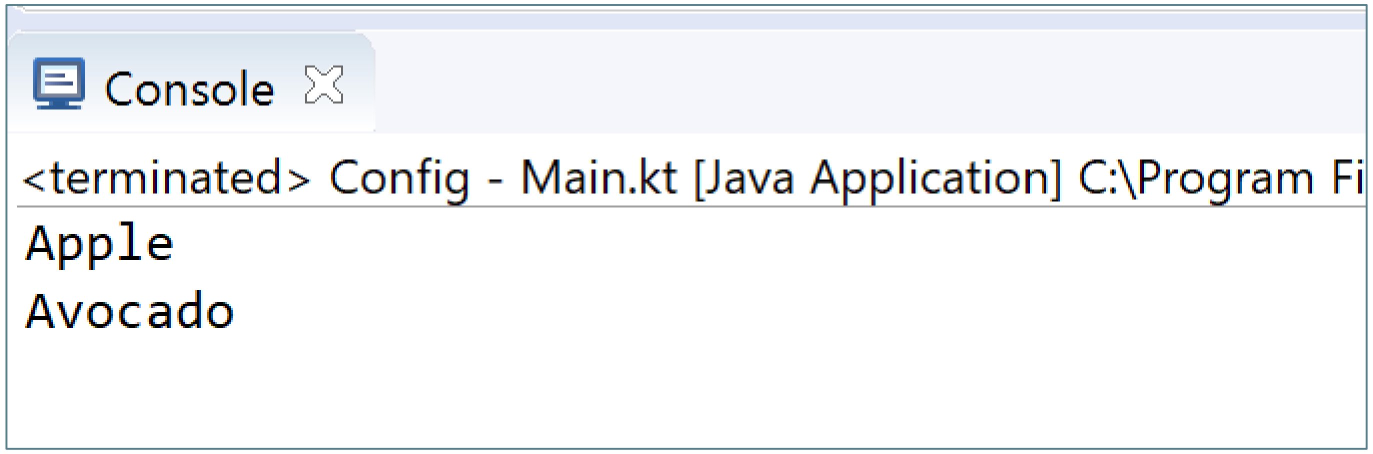


```
<terminated> Config - Main.kt [Java Application] C:\Program Files\Java\jdk1.8.0_
Avocado
Apple
```

Collections – lambdas

Using lambda expressions to filter and map collections

```
fun main(args: Array<String>) {

  val fruits = listOf ("Banana", "Avocado", "Apple", "Kiwi")
  fruits.filter   {it.startsWith("A")
        .sortedBy { it }
        .forEach  {println(it)}
}
```

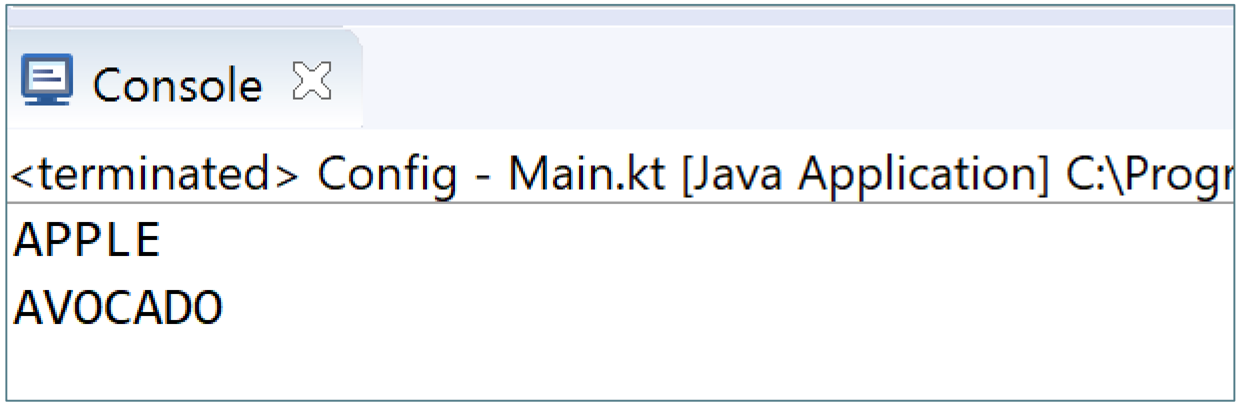


```
<terminated> Config - Main.kt [Java Application] C:\Program Fi
Apple
Avocado
```

Collections – lambdas

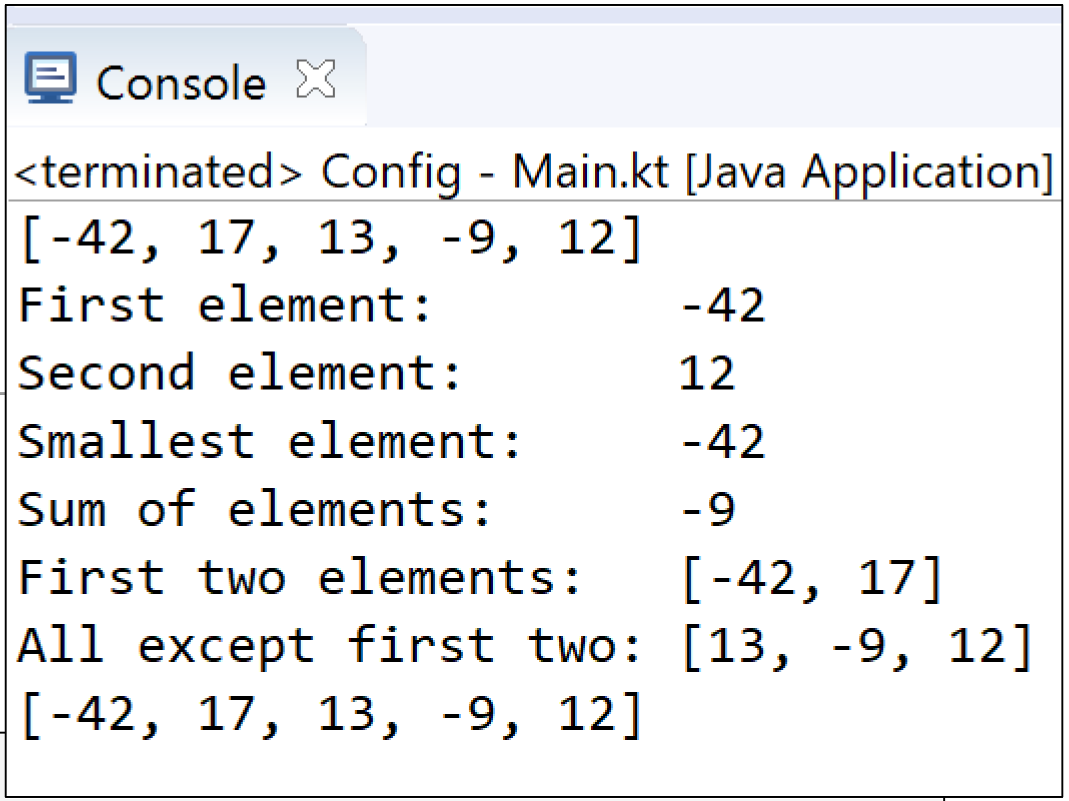
Using lambda expressions to filter and map collections

```
fun main(args: Array<String>) {

  val fruits = listOf ("Banana", "Avocado", "Apple", "Kiwi")
  fruits.filter   {it.startsWith("A")
        .sortedBy { it }
        .map      {it.toUpperCase()}
        .forEach  {println(it)}
}
```



```
<terminated> Config - Main.kt [Java Application] C:\Progr
APPLE
AVOCADO
```

Collections – sample functions

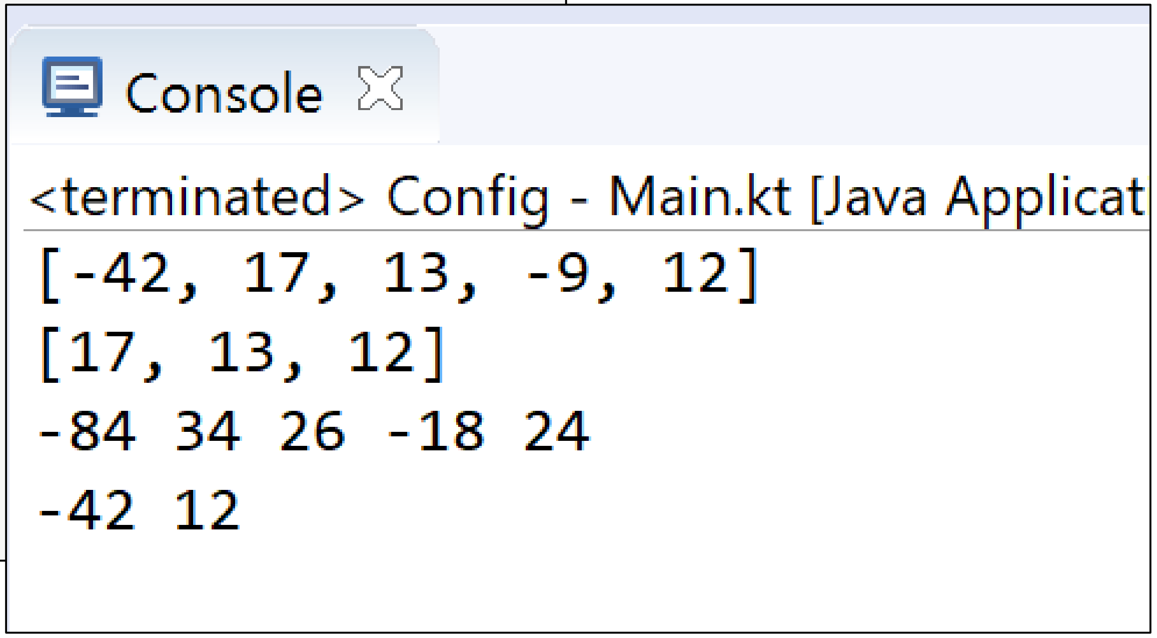


```
<terminated> Config - Main.kt [Java Application]
[-42, 17, 13, -9, 12]
First element:        -42
Second element:       12
Smallest element:     -42
Sum of elements:      -9
First two elements:   [-42, 17]
All except first two: [13, -9, 12]
[-42, 17, 13, -9, 12]
```

```
fun main(args: Array<String>) {

  val numbers = listOf(-42, 17, 13, -9, 12)
  println(numbers)

  println("First element:        " + numbers.first())
  println("Second element:       " + numbers.last())
  println("Smallest element:     " + numbers.min())
  println("Sum of elements:      " + numbers.foldRight
                                   (0, { a, b -> a + b }))
  println("First two elements:   " + numbers.take(2))
  println("All except first two: " + numbers.drop(2))

  println(numbers)
}
```

Collections – sample functions

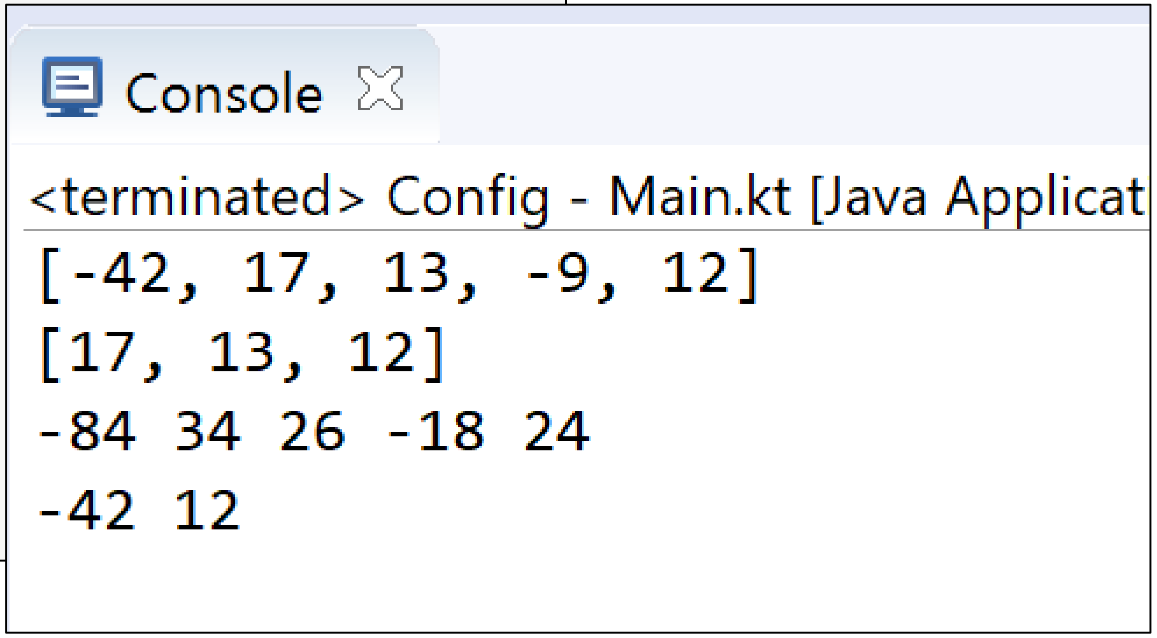
```
fun main(args: Array<String>) {

  val numbers = listOf(-42, 17, 13, -9, 12)
  println(numbers)

  // New list only containing non-negative numbers
  val nonNegative = numbers.filter { it >= 0 }
  println(nonNegative)

  // Double each element
  numbers.forEach { print("${it * 2} ") }
  println();

  // Output Even elements only
  numbers.filter {it % 2 == 0}
         .forEach {print ("$it " )}
  println();

}
```



```
<terminated> Config - Main.kt [Java Applicat
[-42, 17, 13, -9, 12]
[17, 13, 12]
-84 34 26 -18 24
-42 12
```

Sets and Lambdas

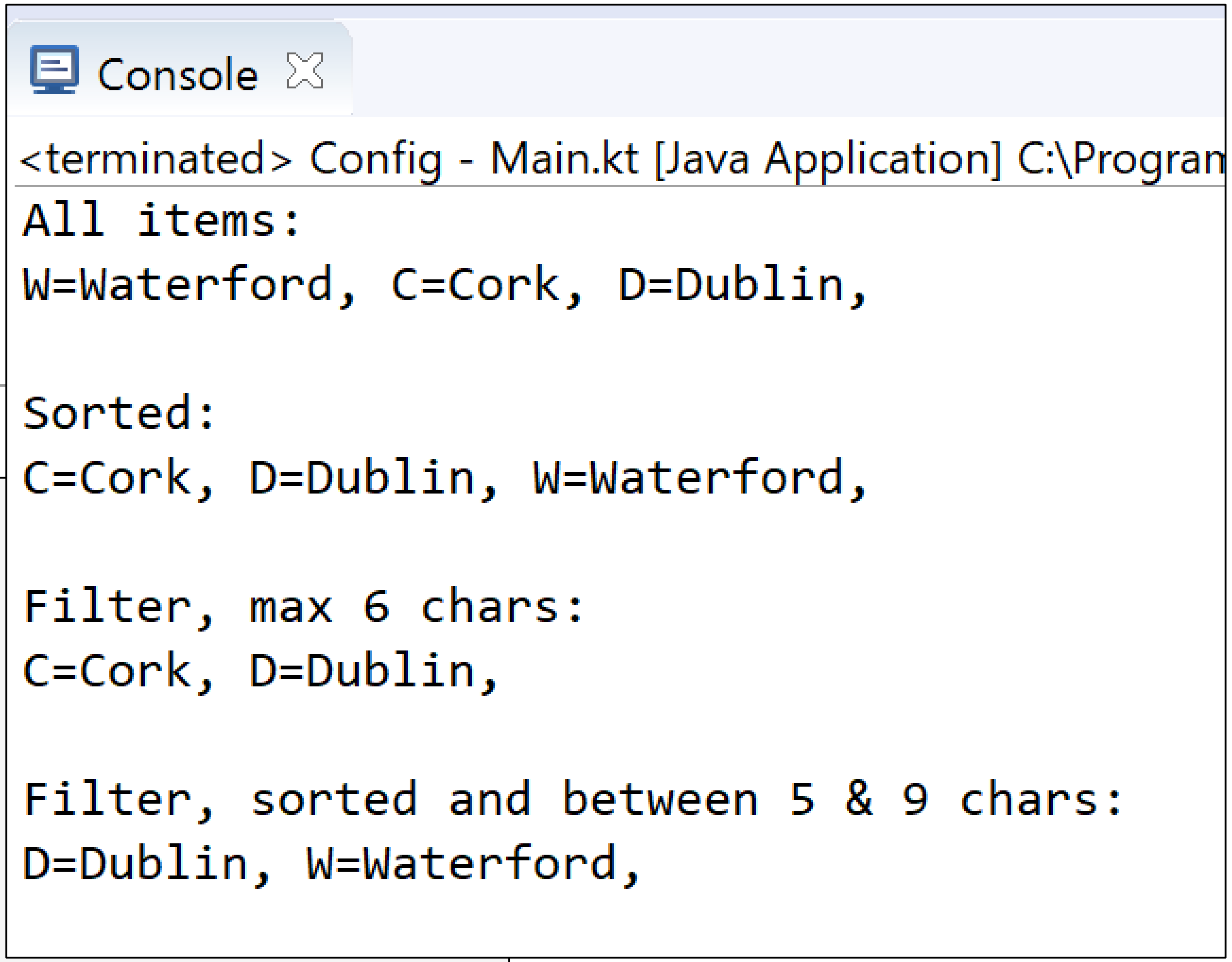
```
fun main(args: Array<String>) {

  val numbers = setOf(-42, 17, 13, -9, 12)
  println(numbers)

  // New list only containing non-negative numbers
  val nonNegative = numbers.filter { it >= 0 }
  println(nonNegative)

  // Double each element
  numbers.forEach { print("${it * 2} ") }
  println();

  // Output Even elements only
  numbers.filter {it % 2 == 0}
         .forEach {print ("$it " )}
  println();

}
```



```
<terminated> Config - Main.kt [Java Applicat
[-42, 17, 13, -9, 12]
[17, 13, 12]
-84 34 26 -18 24
-42 12
```

Maps and Lambdas

```
fun main(args: Array<String>) {

    val counties = mapOf(
            Pair("W","Waterford"),
            Pair("C","Cork"),
            Pair ("D","Dublin"))

    println("All items:");
    counties.forEach {print(it); print (", ")}

    println("\n\nSorted:");
    counties.toSortedMap()
            .forEach {print(it); print (", ")}

    println("\n\nFilter, max 6 chars:");
    counties.filter {it.value.length <= 6 }
            .forEach {print(it); print (", ")}

    println("\n\nFilter, sorted and between 5 & 9 chars:");
    counties.filterValues {it.length >= 5 && it.length <=9}
            .toSortedMap()
            .forEach {print(it); print (", ")}
}
```



```
<terminated> Config - Main.kt [Java Application] C:\Progran
All items:
W=Waterford, C=Cork, D=Dublin,

Sorted:
C=Cork, D=Dublin, W=Waterford,

Filter, max 6 chars:
C=Cork, D=Dublin,

Filter, sorted and between 5 & 9 chars:
D=Dublin, W=Waterford,
```

Arguments

default and named

Default Arguments (optional)

In Java, you often have to duplicate code in order **define different variants of a method or constructor (i.e. overloading)**.

Kotlin simplifies this by using default values for arguments (i.e. makes them optional arguments).

Default Arguments (optional)

```
class NutritionFacts(val foodName: String,
                     val calories: Int,
                     val protein: Int = 0,
                     val carbohydrates: Int = 0,
                     val fat: Int = 0,
                     val description: String = "")
{
}
```

Primary Constructor

Optional Parameters

```
val pizza  = NutritionFacts("Pizza",  442, 12, 27, 24, "Deep Pan Pizza")
val pasta = NutritionFacts("Pasta", 371, 14, 25, 11)
val soup  = NutritionFacts("Soup",  210)
```

Some possible constructor calls

Named Arguments

```
class NutritionFacts(val foodName: String,
                     val calories: Int,
                     val protein: Int = 0,
                     val carbohydrates: Int = 0,
                     val fat: Int = 0,
                     val description: String = "")
{
}
```

```
val pasta  = NutritionFacts("Pasta", 371, 14, 25, 11)
val burger = NutritionFacts("Hamburger", calories = 541, fat = 33, protein = 14)
val rice   = NutritionFacts("Rice", 312, carbohydrates = 23, description = “Grains")
```

Naming arguments make your code more readible

Some additional sources for exploration:

Inheritance	https://www.programiz.com/kotlin-programming/inheritance
Interfaces	https://www.programiz.com/kotlin-programming/interfaces
Collections	https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/index.html
Try examples online	https://try.kotlinlang.org/#/Examples/Hello,%20world!/Simplest%20version/Simplest%20version.kt
Encapsulation & Polymorphism	https://medium.com/@napperley/kotlin-tutorial-12-encapsulation-and-polymorphism-6e5a150f25e1
Spek (testing)	https://objectpartners.com/2016/02/23/an-introduction-to-kotlin/ https://github.com/mike-plummer/KotlinCalendar