НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Algorithms & Programming

(p.5 – Arrays & Lists)

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

What is an array?

- An array is an ordered collection of values of the same type.
- The elements in the array are **zero-indexed**, which means the index of the first element is 0, the index of the second element is 1, and so on.

5	-12	-12	9	10	0	-9	-12	-1	23	65	64	11	43	39	-15
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	a[15]

There are sixteen elements in this array, at indices 0 - 15

All values are of type Int, so you can't add non-Int types to an array that holds integers. Notice that the same value can appear multiple times.

When are arrays useful?

- Arrays are useful when you want to store your items in a particular order.
- You may want the elements sorted, or you may need to fetch elements by index without iterating through the entire array.
- For example, if you were storing high score data, then order would matter.
- You would want the highest score to come first in the list (i.e. at index 0) with the next-highest score after that, and so on.

Creating arrays

- The easiest way to create an array is by using a function from the Kotlin standard library, arrayOf().
- This is a concise way to provide array values.

```
val evenNumbers = arrayOf(2, 4, 6, 8)
```

- Since the array only contains integers, Kotlin infers the type of evenNumbers to be an array of `Int` values.
- This type is written as `Array<Int>`. The type inside the angle brackets defines the type of values the array can store, which the compiler will enforce when you add elements to the array.

Creating arrays

- It's also possible to create an array with all of its values set to a default value:

```
val fiveFives = Array(5) { 5 } // 5, 5, 5, 5, 5
```

- As with any type, it's good practice to declare arrays that aren't going to change as constants using **`val`**. For example, consider this array:

```
val vowels = arrayOf("a", "e", "i", "o", "u")
```

`vowels` is an array of strings and its values can't be changed.

Arrays of primitive types

- The Kotlin standard library contains functions other than arrayOf() that make it possible to create arrays that correspond to arrays of primitive types. For example, you can create an array of odd numbers as follows:

```
val oddNumbers = intArrayOf(1, 3, 5, 7)
```

- When running Kotlin on the JVM, the oddNumbers array is compiled to a Java array of type `int[]`.

Arrays of primitive types

- Other standard library functions include floatArrayOf(), doubleArrayOf(), and booleanArrayOf().
- These various functions create arrays of type IntArray, FloatArray, DoubleArray, etc.
- You can also pass a number into the constructor for these types, for example, to create an array of zeros:

```
val zeros = DoubleArray(4) // 0.0, 0.0, 0.0, 0.0
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Arrays of primitive types

- You can convert between the boxed and primitive arrays using functions like toIntArray()

```
val otherOddNumbers = arrayOf(1, 3, 5, 7).toIntArray()
```

The type of otherOddNumbers is IntArray and not Array<Int>

Lists

- Arrays are typically more efficient than lists in terms of raw performance, but lists have the additional feature of being **dynamically-sized**.
- That is, arrays are of fixed-size, but lists can be setup to grow and shrink as needed

Creating lists

- Like with arrays, the standard library has a function to create a list.

```
val innerPlanets = listOf(
    "Mercury",
    "Venus",
    "Earth",
    "Mars")
```

- The type of `innerPlanets` is inferred to be `List<String>`, with `String` being another example of a type argument.
- So `innerPlanets` can be passed into any function that needs a `List`.

Creating lists

- If, for some reason, you explicitly want innerPlanets to have the type ArrayList, there is a different standard library function you can use

```
val innerPlanetsArrayList = arrayListOf(
    "Mercury",
    "Venus",
    "Earth",
    "Mars")
```

- innerPlanets can’t be altered once created.
- but innerPlanetsArrayList can
- An empty list can be created by passing no arguments into list()

Creating lists

- Because the compiler isn't able to infer a type from this, you need to use a type declaration to make the type explicit:

```
val subscribers: List<String> = listOf()
```

- You could also put the type argument on the function:

```
val subscribers = listOf<String>()
```

- Since the list returned from listOf() is immutable, you won't be able to do much with this empty list.
- Empty lists become more useful as a starting point for a list when they're **mutable**.

Mutable lists

- Once again, the standard library has a function to use here.

```
val outerPlanets = mutableListOf(
    "Jupiter",
    "Saturn",
    "Uranus",
    "Neptune")
```

You've made outerPlanets a mutable list, just in case Planet X is ever discovered in the outer solar system. You can create an empty mutable list by passing no arguments to the function:

```
val exoPlanets = mutableListOf<String>()
```

Accessing elements

- Being able to create arrays and lists is useless unless you know how to fetch values from them.
- The syntax is similar for both arrays and lists.

Using properties and methods

- Imagine you're creating a game of cards, and you want to store the players' names in a list.
- The list will need to change as players join or leave the game, so you need to declare a mutable list:

```
val players = mutableListOf(
    "Alice",
    "Bob",
    "Cindy",
    "Dan")
```

Using properties and methods

- Before the game starts, you need to make sure there are enough players. You can use the isEmpty() *method* to check if there's at least one player:

```
print(players.isEmpty())
// > false
```

The list isn't empty, but you need at least two players to start a game. You can get the number of players using the size *property*:

```
if (players.size < 2) {
    println("We need at least two players!")
} else {
    println("Let's start!")
}
// > Let's start!
```

Using properties and methods

- How would you get the first player's name?
- Lists provide the first() method to fetch the first object of a list:

```
var currentPlayer = players.first()
println(currentPlayer)                    // > Alice
```

If list is empty it will throw an exception.
Similarly, lists have a last() method that returns the last value in a list, or throws an exception if the list is empty:

```
println(players.last()) // > Dan
```

Using properties and methods

- If the array contained strings, then it would return the string that's the lowest in alphabetical order, which in this case is "Alice":

```
// val minPlayer = players.min()      // before 1.4
val minPlayer = players.minOrNull()   // since  1.4
println("$minPlayer will start")
```

- Instead of throwing an exception if no minimum can be determined, `min()` returns a nullable type, so you need to check if the value returned is `null`
- So, in new versions of Kotlin, it was deprecated.
- You should to use `minOrNull()` instead

Using properties and methods

- Obviously, first() and minOrNull() will not always return the same value. For example:

```
println(arrayOf(2, 3, 1).first())
// > 2
println(arrayOf(2, 3, 1).minOrNull())
// > 1
```

As you might have guessed, lists also have a maxOrNull() method (simple max() – before 1.4)

```
val maxPlayer = players.maxOrNull()
if (maxPlayer != null) {
    println("$maxPlayer is the MAX") // > Dan is the MAX
}
```

Using indexing

- The most convenient way to access elements in an array or list is by using the indexing syntax.
- This syntax lets you access any value directly by using its index inside square brackets:

```
val firstPlayer = players[0]
println("First player is $firstPlayer")
// > First player is Alice
```

Because arrays and lists are zero-indexed, you use index 0 to fetch the first object.

Using indexing

- You can use a greater index to get the next elements in the array or list, but if you try to access an index that's beyond the size of the array or list, you'll get a runtime error.

```
val player = players[4] // > IndexOutOfBoundsException
```

- You receive this error because players contains only four strings.
- Index 4 represents the fifth element, but there *is* no fifth element in this list.

Using ranges to slice

- You can use the slice() method with ranges to fetch more than a single value from an array or list.
- For example, if you'd like to get the next two players, you could do this:

```
val upcomingPlayersSlice = players.slice(1..2)
println(upcomingPlayersSlice.joinToString())
// > Bob, Cindy
```

The object returned from the slice() method is a separate array or list from the original, so making modifications to the slice does not affect the original array or list.

Checking for an element

- You can check if there's at least one occurrence of a specific element by using the in operator, which returns true if it finds the element, and false otherwise.
- You can use this strategy to write a function that checks if a given player is in the game:

```
fun isEliminated(player: String): Boolean {
    return player !in players
}
```

```
println(isEliminated("Bob")) // > false
```

```
players.slice(1..3).contains("Alice") // false
```

Modifying lists

- You can make all kinds of changes to mutable lists, such as adding and removing elements, updating existing values, and moving elements around into a different order.
- Now we'll see how to work with the list to match up with what's going on in your game.

Appending elements

- If new players want to join the game, they need to sign up and add their names to the list.
- Eli is the first player to join the existing four players. You can add Eli to the end of the array using the add() method:

```
players.add("Eli")
```

The next player to join the game is Gina. You can add her to the game another way, by using the += operator:

```
players += "Gina"
```

Appending elements

- While arrays are of fixed-size, you *can* in fact use the += operator with an array that is declared as var

```
var array = arrayOf(1, 2, 3)
array += 4
println(array.joinToString()) // > 1, 2, 3, 4
```

But beware that you are not actually appending the value onto the existing array, but instead creating an entirely new array that has the additional element and assigning the new array to the original variable.

Inserting elements

- You want to add player with name Frank to the list between Eli and Gina.
- To do that, you can use a variant of the add() method that accepts an index as the first argument:

```
players.add(5, "Frank")
```

The first argument defines where you want to add the element. Remember that the list is zero-indexed, so index 5 is Gina's index, causing her to move up as Frank takes her place.

Removing elements

- During the game, the other players caught Cindy and Gina cheating. They should be removed from the game!
- You can remove them by name using the remove() method:

```
val wasPlayerRemoved = players.remove("Gina")
println("It is $wasPlayerRemoved that Gina was removed")
// > It is true that Gina was removed
```

This method does two things: It removes the element and then returns a Boolean indicating whether the removal was successful, so that you can make sure the cheater has been removed!

Removing elements

- To remove Cindy from the game, you need to know the exact index where her name is stored.
- Looking at the list of players, you see that she's third in the list, so her index is 2.
- You can remove Cindy using removeAt()

```
val removedPlayer = players.removeAt(2)
println("$removedPlayer was removed")
// > Cindy was removed
```

Unlike remove(), removeAt() returns the element that was removed from the list.
You could then add that element to a list of cheaters!

Finding element

- But how would you get the index of an element if you didn't already know it?
- There's a method for that! indexOf() returns the *first index* of the element, because the list might contain multiple copies of the same value.
- If the method doesn't find the element, it returns -1.

```
val indexOfDan = players.indexOf("Dan”)
```

Updating elements

- Frank has decided everyone should call him Franklin from now on. You could remove the value "Frank" from the list and then add "Franklin", but that's too much work for a simple task.
- Instead, you should use the indexing syntax to update the name.

```
println(players.joinToString())
// > "Alice", "Bob", "Dan", "Eli", "Frank"
players[4] = "Franklin"
println(players.joinToString())
// > "Alice", "Bob", "Dan", "Eli", "Franklin"
```

Be careful to not use an index beyond the bounds of the list, or your code will crash.

Updating elements

- As the game continues, some players are eliminated, and new ones come to replace them.
- You can use indexing to replace the old players with the new:

```
players[3] = "Anna"
players.sort()
println(players.joinToString())
// > "Alice", "Anna", Bob", "Dan", "Franklin
```

This code replaces the player Eli with the player Alice. You then call sort() on the list to make sure the list remains sorted in alphabetical order.

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Iterating through a list

- Before the players leave, you want to print the names of those still in the game.
- Like for arrays, you can do this using the for loop

```
for (player in players) {
     println(player)
}
// > Alice
// > Anna
// > Bob
// > Dan
// > Franklin
```

Iterating through a list

- If you need the index of each element, you can iterate over the return value of the list's withIndex() method, which can be *destructed* to each element's index and value:

```
for ((index, player) in players.withIndex()) {
    println("${index + 1}. $player")
}
// > 1. Alice
// > 2. Anna
// > 3. Bob
// > 4. Dan
// > 5. Franklin
```

Iterating through a list

- It’s getting late, so the players decide to stop for the night and continue tomorrow.
- In the meantime, you’ll keep their scores in a separate list.

```
val scores = listOf(2, 2, 8, 6, 1)
```

Iterating through a list

```
val scores = listOf(2, 2, 8, 6, 1)
```

Now you can use the technique you've just learned to write a function that takes a list of integers as its input and returns the sum of its elements:

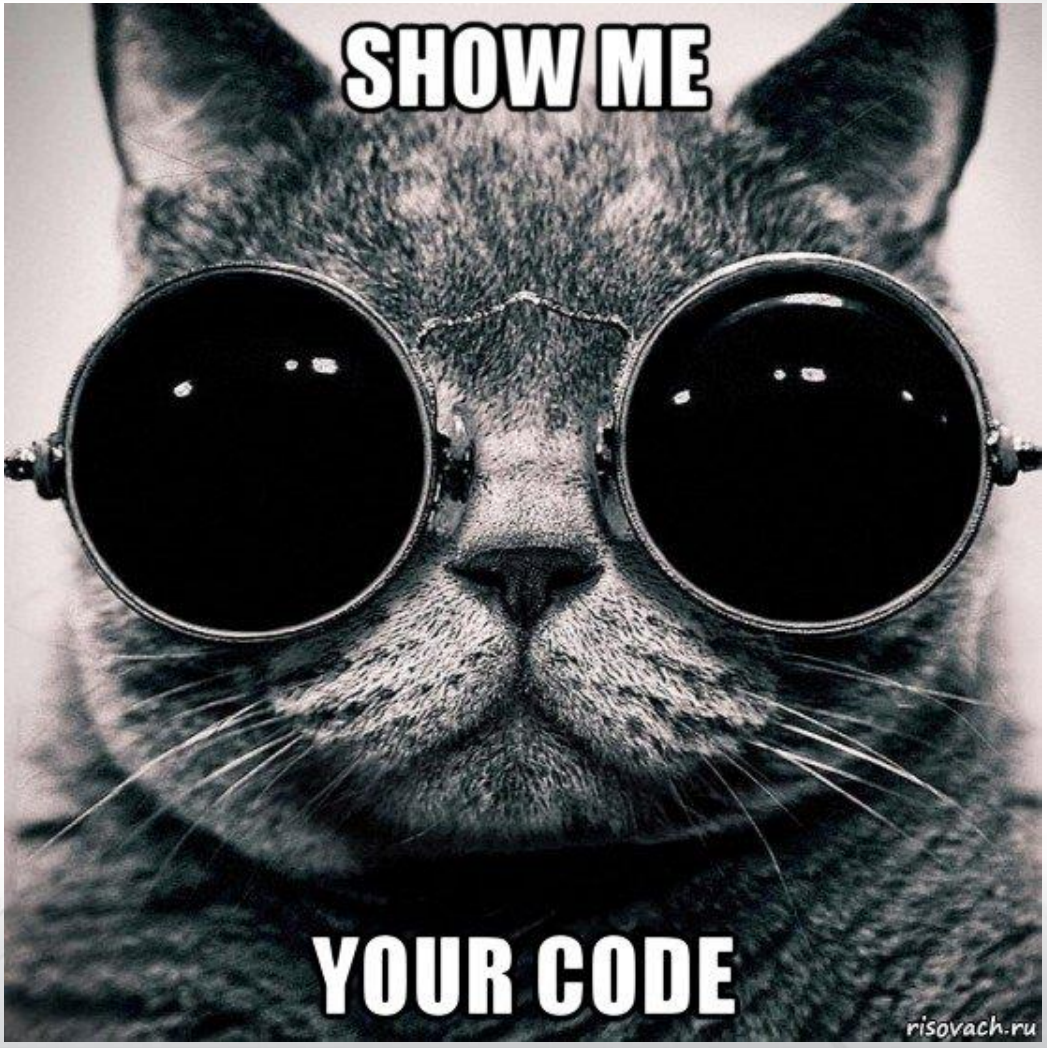
```
fun sumOfElements(list: List<Int>): Int {
    var sum = 0
    for (number in list) {
        sum += number
    }
    return sum
}
```

You could use this function to calculate the sum of the players' scores:

```
println(sumOfElements(scores)) // > 19
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Let’s code!



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Questions?

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ КОРАБЛЕБУДУВАННЯ ІМЕНІ АДМІРАЛА МАКАРОВА

Algorithms & Programming

(p.5 – Arrays & Lists)

Yevhen Berkunskyi, NUoS
eugeny.berkunsky@gmail.com
http://www.berkut.mk.ua