

Lists

Learn how to create and use lists in Kotlin to store and manipulate multiple values in one data structure.

WE'LL COVER THE FOLLOWING



- Creating a List
- Read-only vs Mutable Lists
 - Why immutability?
- Working with Lists
 - Accessing Elements
 - Manipulating Elements
 - Overview: Kotlin's Collection Types
- Quiz
- Exercises
- Summary

Collections are a central feature of any programming language. They allow storing multiple values – such as your list of favorite books – in one convenient data structure which can then be stored and accessed with a single variable.

The fundamental collection types you'll master in the following lessons are:

- Lists
- Sets
- Arrays*
- Maps

*Arrays are not part of the actual Collections API but are included here because they're also a data structure for storing multiple elements.

Creating a List

Creating a List

Kotlin provides three convenient functions to initialize lists:

```
val siUnits = listOf("s", "m", "kg", "A", "K", "mol", "cd") // Creates read-only list
val quarks = mutableListOf("up", "down", "charm", "strange", "top", "bottom") // Creates mutable list
val physicists = arrayListOf("Albert Einstein", "Isaac Newton") // Creates mutable list
```



All three functions allow you to add any number of elements (try it out!). The main difference between them is that `listOf` creates a **read-only list** whereas `mutableList` creates a **mutable list**. What does this mean? With read-only lists, you cannot add or remove elements from the list after it's created. With a mutable list, however, you can.

The code example uses type inference so you can't see what the data types are. They are:

Function Call	Return Type
<code>listOf("", ...)</code>	<code>List<String></code>
<code>mutableListOf("", ...)</code>	<code>MutableList<String></code>
<code>arrayListOf("", ...)</code>	<code>ArrayList<String></code>

You can read the type `List<String>` as “list of string”.

Mini-exercise: add explicit types to the code widget above and run it again.

Read-only vs Mutable Lists

The differentiation between read-only lists and mutable lists has the same underlying intention as differentiating between `val` vs `var`: immutability. At this point it is important to understand that `List` and `MutableList` are not

this point, it's important to understand how `val` and read-only data structures differ. They essentially represent two levels of immutability:

- `val` prevents an initialized variable from being reassigned to another value. In other words, the memory address the variable points at cannot be changed; it will always point to the same memory address.
- Read-only collections prevent adding or removing elements from an existing collection but don't affect any variables that may point to these collections. In other words, they disallow changing values in the section of the computer's memory that stores the collection.

So how do these two play together?

	<code>val</code>	<code>var</code>
Read-only collection	Cannot reassign variable or change values stored in variable ("immutable")	Can reassign the variable (to a whole different collection) but can't overwrite the values in the original collection
Mutable collection	Cannot reassign variable but can freely change the values stored in the variable	Can both reassign the variable and change values in the collection it stores

Following the principle of immutability, use read-only collections and `val` s whenever you can.

Why immutability?

Using immutable data structures and `val` s allow you to better understand the data flow of your program. This facilitates enforcing invariants in your code, since you know that the contents of your variables cannot change. First, this avoids a variety of hard-to-fix bugs.

Second, immutability is a fundamental principle of functional programming, which is a widely popular programming pattern.

Note that Kotlin's collections are not strictly immutable because there *are* intricate ways you could use to manipulate values in them. However, the Kotlin team is working on a library of actual immutable collections as opposed to just read-only collections.

Working with Lists

To work with lists, you must be able to access and manipulate its individual elements.

Accessing Elements

To access an element from a list, you use Kotlin's **indexed access operator** using square brackets:

```
val siUnit = siUnit[2]           // Gets 3rd element of siUnits
val quark = quarks[0]            // Gets 1st element of quarks
// val physicist = physicists[3] // Would cause an error because there is no 4th element in
```



As you probably noticed, indexing starts at zero. So passing in i gives you the $i+1$ -th element of a list. Consequently, the maximum index you can use without causing a runtime error is $i-1$.

Manipulating Elements

To manipulate the elements of a list, you also use the indexed access operator and reassign the value:

```
// siUnit[1] = "meter"           // Would cause error because siUnits is readonly
quarks[2] = "CHARM"              // Overwrites 3rd element of quarks
physicists[1] = "Marie Curie"    // Overwrites 2nd element of physicists
```



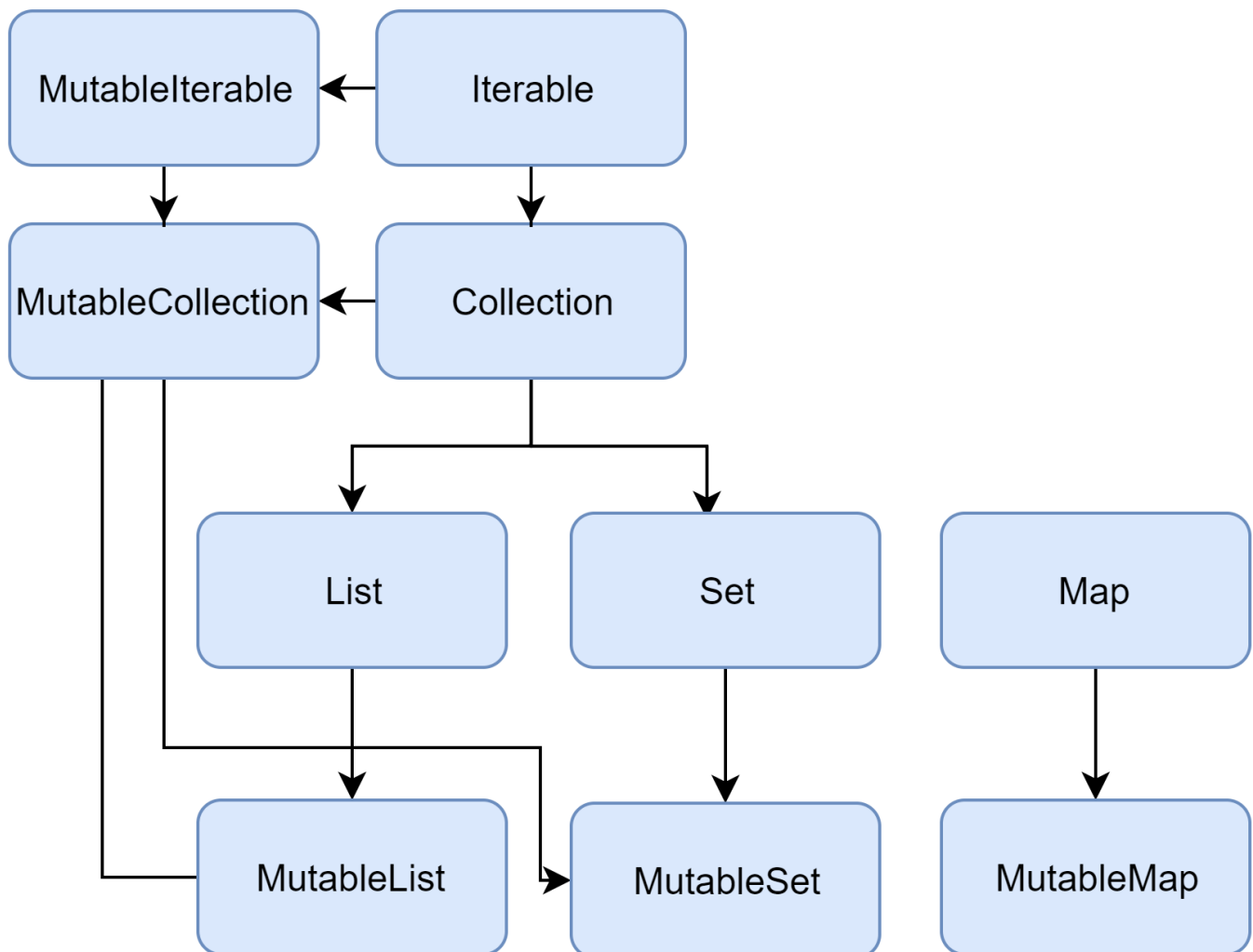
The syntax here is the same as for regular variable assignments, except for the additional index access in square brackets. Naturally, trying to overwrite a value of a read-only collection causes a compile-time error.

Compile-time error means that Kotlin won't allow the code to run. The compiler knows that it would fail, so it will reject compiling this program. It

might not appear so, but compile-time errors are your friend; they prevent nasty runtime errors that may require long debugging or, even worse, could crash a productive application. Kotlin's compiler is very good at catching a variety of potential mistakes.

Overview: Kotlin's Collection Types

In Kotlin, every collection type has a mutable and read-only variant, as portrayed in the following diagram:



Kotlin collections inheritance hierarchy. Each collection type has a readonly and a mutable variant.
[Source: <https://kotlinlang.org/docs/reference/collections-overview.html>]

`Iterable` and `Collection` are the more abstract interfaces from which lists and maps inherit. As you can see, `MutableList` is both a `List` and a `MutableCollection`, as indicated by the arrows. `Map`, which we'll cover later, doesn't inherit from `Collection` itself but is still part of the Collections API.

Note: Arrays are not part of the Kotlin Collections API, so they are not

Note: Arrays are not part of the Kotlin Collections API, so they are not included in this diagram. Arrays also do not have a read-only variant.

Quiz

Mutable and readonly lists

1

Which of the following create *mutable* lists? You can select multiple answers.

COMPLETED 0%

1 of 4



Exercises

In the following code widget, add three variables (use the highlighted variable names):


1. A read-only list of your best `childhoodFriends`
2. A mutable list of `books` you read in the last 6 months


After that, extend your code to complete the following challenges:

- Overwrite the 2nd favorite book in your list with any other book title
- If your list of books has 10 or more elements, print `"I devour books"`. If it has at least 5 elements, print `"I enjoy reading books from time to time"`.





Otherwise, print "I'm more of a sporadic reader of books".

- Which condition construct is best for this use case?

 Problem

 Solution

// Add your code here



Collections are fundamental in programming and you will use them again in future exercises. This will help you solidify your knowledge of collections.

Summary

Like any higher-level programming language, Kotlin offers collections as data structures to store multiple values.

- What makes Kotlin different is that it clearly separates read-only and mutable collections.
- Lists can contain any number of elements. In mutable lists, elements can be changed, added, or removed.
- Particular elements of a list are accessed by index, using square brackets.

In the next lesson, you will learn about sets as an alternative data structure to store multiple elements.