Using List

We'll cover the following

Creating a list in Kotlin

As a first step in creating a list, Kotlin wants you to declare your intent—immutable or mutable. To create an immutable list, use `listOf()`—immutability is implied, which should also be our preference when there's a choice. But if you really need to create a mutable list, then use `mutableListOf()`.

The function `listOf()` returns a reference to an interface `kotlin.collections.List<T>`. In the following code the reference fruits is of this interface type, specialized to String for the parametric type:

```
val fruits: List<String> = listOf("Apple", "Banana", "Grape")
println(fruits) //[Apple, Banana, Grape]
```

lists.kts

To access an element in the list you may use the traditional `get()` method, but the index operator `[]`, which routes to the same method, may be used as well.

```
// lists.kts
println("first's ${fruits[0]}, that's ${fruits.get(0)}")
  //first's Apple, that's Apple
```

The index operator `[]` is less noisy than `get()` and is more convenient—use it freely instead of `get()`. You may check if a value exists in the collection using the `contains()` method or using the in operator—we'll dig into operator overloading in Overloading Operators.

in Overloading Operators.

```
// lists.kts
println(fruits.contains("Apple")) //true
println("Apple" in fruits) //true
```

In the previous code we asked if the list contains a value and if that value is in the list—the latter is more fluent and is preferred in Kotlin.

Using the reference returned by `listOf()`, we can't modify the list. If you're the good type that won't take no for an answer, go ahead and verify that with the following code:

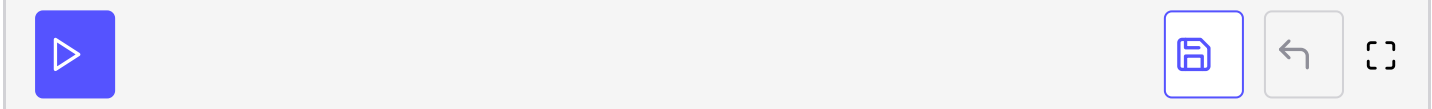
```
// lists.kts
fruits.add("Orange") //ERROR: unresolved reference: add
```

The interface `kotlin.collections.List<T>` acts as a compile-time view around the widely used object in the JDK that you'd create using `Arrays.asList()` in Java, but the interface doesn't have the methods that permit mutation or change to the list. That's the reason the call to the `add()` method failed at compile time. By providing such views, Kotlin is able to make the code safer from the immutability point of view, but without introducing any runtime overhead or conversions.

That protection is nice, but that shouldn't stop us from having another fruit. This is where the `+` operator comes in handy.

```
val fruits: List<String> = listOf("Apple", "Banana", "Grape")
val fruits2 = fruits + "Orange"

println(fruits) //[Apple, Banana, Grape]
println(fruits2) //[Apple, Banana, Grape, Orange]
```



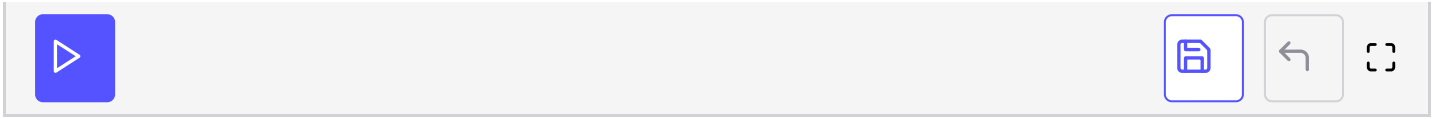
lists.kts

The operation didn't mutate the list `fruits`; instead, it created a new list with all the values from the original copied over, plus the new element.

If there's a plus, then it's only logical to expect a minus. The `-` operator is useful to create a new list without the first occurrence of the specified element, like this:

```
val fruits: List<String> = listOf("Apple", "Banana", "Grape")
val noBanana = fruits - "Banana"


println(noBanana) //[Apple, Grape]
```



lists.kts

If the specified element isn't present in the list, then the result is a list with the original elements, nothing removed.

The List interface shines nicely in the previous examples, and Kotlin provides a long list of methods. The `fruits` interface is of type `List<T>`, but what's the real class under the hood? you may wonder. Let's answer that question with the next piece of code:

```
// lists.kts
println(fruits::class) //class java.util.Arrays$ArrayList
println(fruits.javaClass) //class java.util.Arrays$ArrayList
```

The output shows that the instance is of the type provided in the JDK, even though we accessed it using a view interface in Kotlin.

Mutable lists

The `listOf()` method returns a read-only reference, but if you feel the urge to create a mutable list, call a helpline so they can convince you not to. But, after enough thought and discussions, if you decide that's the right choice, you can create one using the `mutableListOf()` function. All the operations you were able to perform on `List<T>` are readily available on the instance of `MutableList<T>` as well. The instance created using this method, though, is an instance of `java.util.ArrayList` instead of `java.util.Arrays$ArrayList`.

In the next piece of code, we get access to the read-write interface instead of the read-only interface:

```
val fruits: MutableList<String> = mutableListOf("Apple", "Banana", "Grape")
println(fruits::class) //class java.util.ArrayList
```

Using this interface we may alter the list. For example, we can add an element to it:

```
fruits.add("Orange")
```

Instead of interacting with the `ArrayList<T>` through the `MutableList<T>` interface, obtained using the `mutableListOf()` function, you may directly obtain a reference of type `ArrayList<T>` using the `arrayListOf()` function.

Where possible, use `listOf()` instead of `mutableListOf()` and `arrayListOf()`—only reluctantly bring in mutability.

Once you create a list, you may iterate over it using the imperative style like we saw in Iterating over Arrays and Lists, or using the functional style like we'll see in Chapter 12, Internal Iteration and Lazy Evaluation.

QUIZ

1 Which type of list should be preferred?

2 Which of the following commands creates an immutable list?

Retake Quiz

If you don’t care about ordering of the elements, you may want to use a set covered in the next lesson instead of a list, and Kotlin has top-level functions for that.