# Internal Iterators

We'll cover the following



We mostly use `for` to program external iterators. But internal iteration involves many specialized tools like `filter()`, `map()`, `flatMap()`, `reduce()`, and so on. Much like the way a professional mechanic uses different specialized tools to fix a car, and doesn't settle for just a hammer, in functional programming we use a combination of the right tools for different tasks. The Kotlin standard library provides plenty of higher-order functions for internal iteration. We'll visit some of the most commonly used functions.

## filter, map, and reduce #

`filter()`, `map()`, and `reduce()` are the three amigos of functional programming; they are fundamental functions used as internal iterators. The `filter()` function picks certain values from a given collection while dropping the others. The `map()` function transforms the values in a collection using a given function or lambda. Finally, the `reduce()` function performs a cumulative operation on the elements, often to arrive at a single value. All these functions perform their operations without mutating or changing the given collection—they return a copy with the appropriate values.

The size of the collection returned by `filter()` may vary from `0` to `n` where `n` is the number of elements in the original collection. The result is a sub-collection; that is, the values in the output collection are values present in the original collection. The lambda passed to `filter()` is applied on each element in the original collection. If and only if the lambda returns `true` when evaluated for an

original collection. If and only if the lambda returns `true`, which evaluated for an

element, the element from the original collection is included in the output collection.

The size of the collection returned by `map()` is the same as the original collection. The lambda passed to `map()` is applied on each element in the original collection, and the result is a collection of these transformed values.

A lambda passed to both `filter()` and `map()` takes only one parameter, but a lambda passed to `reduce()` takes two parameters. The first is an accumulated value and the second is an element from the original collection. The result of the lambda is the new accumulated value. The result of `reduce()` is the result of the last invocation of the lambda.

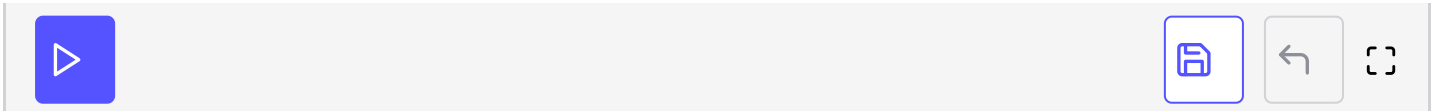
An example will help to illustrate the behavior and purpose of these three functions. For that, let's start with a `Person` class and a `people` collection with some sample values.

```
data class Person(val firstName: String, val age: Int)

val people = listOf(
  Person("Sara", 12),
  Person("Jill", 51),
  Person("Paula", 23),
  Person("Paul", 25),
  Person("Mani", 12),
  Person("Jack", 70),
  Person("Sue", 10))
```

iterators.kts

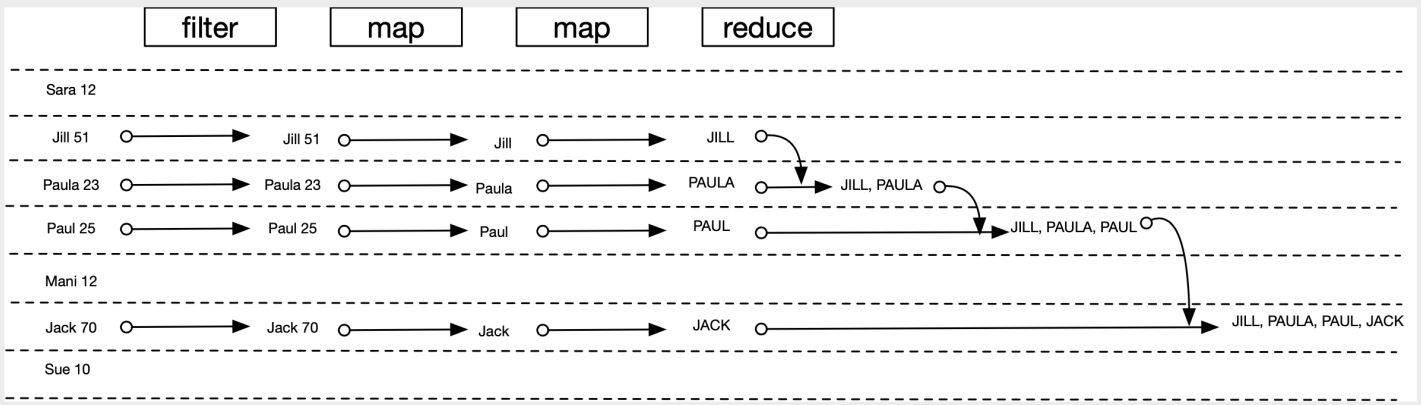
Let's use internal iterators to create the names, in uppercase and comma separated, of everyone who is older than `20`.

```
val result = people.filter { person -> person.age > 20 }
.map { person -> person.firstName }
.map { name -> name.toUpperCase() }
.reduce { names, name -> "$names, $name" }

println(result) //JILL, PAULA, PAUL, JACK
```



iterators.kts

The `filter()` function extracts from the given collection only `Persons` who are older than `20`. That list is then passed on to `map()`, which then transforms the list of `Persons` who are older than `20` to a list of names. The second `map()` then transforms the names list into a list of names in uppercase. Even though we could have combined the two `map()` calls into one, keeping them separate makes the code more cohesive, where each line focuses on one operation. Finally, we combine the uppercase names into one string, comma separated, using the `reduce()` function. The figure below illustrates the operations in the example.



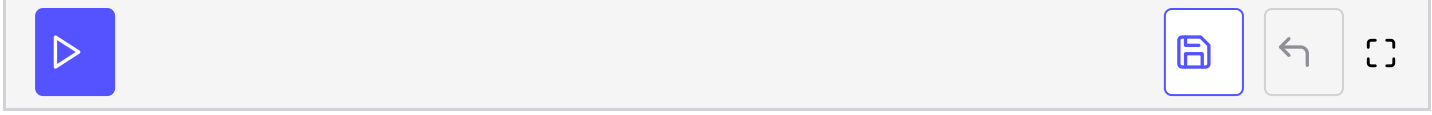

The `filter()` and `map()` functions operate within their swim lanes, where their lambdas return a value based only on the respective elements in the collection, whereas the lambda passed to `reduce()` operates by cutting across the swim lanes. The lambda combines the result of computation for previous elements in the collection with the operation on the subsequent element.

Kotlin provides a number of specialized reduce functions for different operations like sum, max, and even to join strings. We can replace the previous `reduce()` call with the following, to make the code more concise:

```
// iterators.kts
.joinToString(", ")
```

If we want to total the age of every `Person` in the list instead, we can use `map()` and `reduce()`, like so:

```
val totalAge = people.map { person -> person.age }
    .reduce { total, age -> total + age }

println(totalAge) //203
```



iterators.kts

Again, instead of `reduce()`, we may use the specialized reduce operation `sum()`, like so:

```
// iterators.kts
val totalAge2 = people.map { person -> person.age }
  .sum()
```

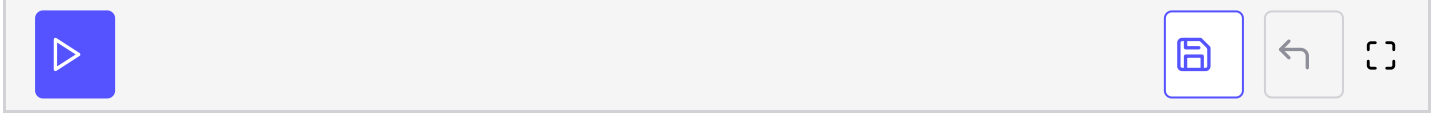
Whereas `reduce()` is a more general cumulative operation on the values in a collection, specialized functions are available for some operations like join and sum. Use the specialized functions where they're available as that makes the code more expressive, less error prone, and easier to maintain as well.

## Getting the first and the last #

Much like the specialized reduce operation `sum()` Kotlin also has a function `first()` to return the first element from a given collection. When used with `filter()` and `map()`, we can perform filtering and transformation before extracting the first element from the resulting collection.

For example, using the `first()` function, let's get the name of the first adult, where adulthood is defined based on the age being greater than `17` rather than the maturity of a person:

```
val nameOfFirstAdult = people.filter { person -> person.age > 17 }
    .map { person -> person.firstName }
    .first()

println(nameOfFirstAdult) //Jill
```



iterators.kts

The `filter()` function returns a collection of everyone who is older than `17`, and the `map()` function returns the names of those adults. Finally, the `first()` function returns the first element from that list of the names of the adults.

If instead of the first adult's name you want to get the last adult's name, replace the

call to `first()` with `last()`. This will result in “Jack” in the above example.

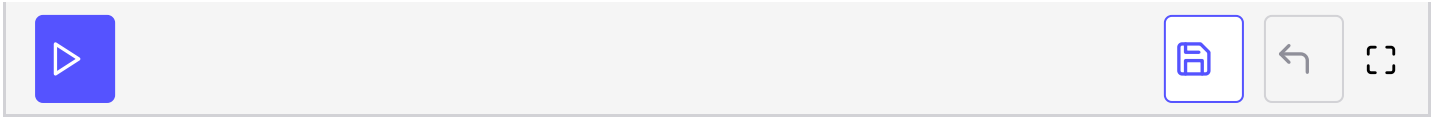
## flatten and flatMap #

Suppose we have a nested list, such as `List<List<Person>>`, where the top-level list contains families and the members of the families are in sublists of `Persons`. What if we want to convert it to one flat list of `Persons`? Kotlin, like languages such as Ruby, has a `flatten()` function for that.

Given an `Iterable<Iterable<T>>` the `flatten()` function will return a `Iterable<T>` where all the elements in the nested iterables are combined into the top level, thus flattening the hierarchy.

Let’s use `flatten()` in a short example.

```
val families = listOf(
    listOf(Person("Jack", 40), Person("Jill", 40)),
    listOf(Person("Eve", 18), Person("Adam", 18)))

println(families.size) //2
println(families.flatten().size) //4
```



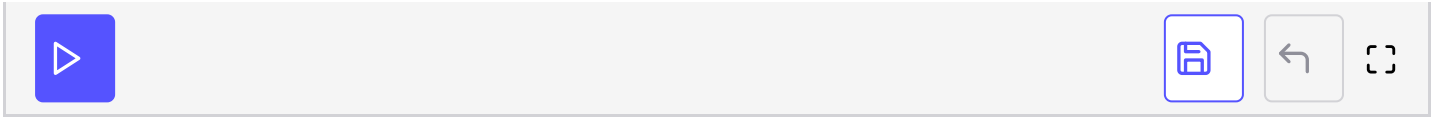
iterators.kts

The variable `families` refers to a nested list of `Person` objects. A call to `size` property on families tells us there are `2` lists contained inside of the outer list. A call to `flatten()` returns a new list which has at the top level the four elements in the nested list.

In the previous example we intentionally created nested lists within a list. Sometimes such nesting may be the result of a `map()` operation on another collection. Let’s explore one such scenario and see how `flatten()` plays a role in that context.

Let’s revisit the `people` collection and get in lowercase the first name and the reverse of the first name for each person. From the `people` collection, we can get the list of first names using a call to the `map()` function. From that, we can get the first names in lowercase again using another call to `map()`. Finally, we can use `map()` a third time to get the name in lowercase and its reverse. Let’s start with these steps and observe the result.

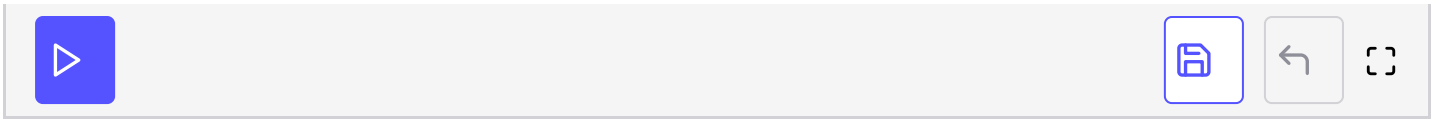
```
val namesAndReversed = people.map { person -> person.firstName }
    .map(String::toLowerCase)
    .map { name -> listOf(name, name.reversed())}

println(namesAndReversed.size) //7
```



iterators.kts

In the last step we returned a list of two strings for each `Person` in the original list. The type of `namesAndReversed` is `List<List<String>>` and the size of the result is `7`, which is the number of elements in the original list. But instead of `List<List<String>>`, we really want `List<String>`. That can be achieved readily with a call to `flatten()`. Let's verify that works.

```
val namesAndReversed2 = people.map { person -> person.firstName }
    .map(String::toLowerCase)
    .map { name -> listOf(name, name.reversed())}
    .flatten()

println(namesAndReversed2.size) //14
```



iterators.kts

The type of `namesAndReversed2` is `List<String>` and the number of elements in it is `14`, as expected. Although that worked, it will be nice if we can combine the map operation with the flatten operation because our intention is to create one flat list and not a nested list. Thus, it would be great if there were a map-flatten function.

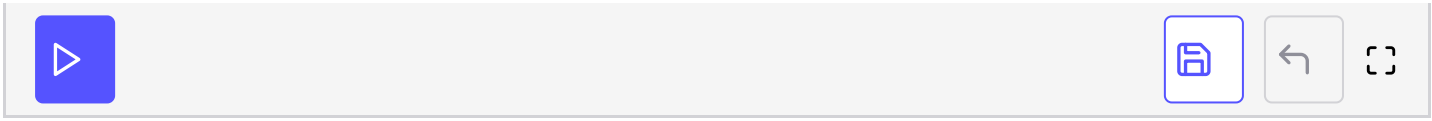
Before we explore this idea further let's work together on a small verbal exercise. Say this aloud three or four times: *map-flatten*.

That resulted in a rather awkward movement of the jaw and a noticeable discomfort. Now imagine there was a function called map-flatten and a generation of programmers grew up saying that name. This may have resulted in the evolution of a species with a weird-shaped jaw to accommodate the odd vocal movement. The designers of functional programming saved the human race by naming it `flatMap()`, even though the actual operation is map followed by a

flattening operation.

Let's combine the last two steps in the previous code into one call to `flatMap():`

```
val namesAndReversed3 = people.map { person -> person.firstName }
    .map(String::toLowerCase)
    .flatMap { name -> listOf(name, name.reversed())}

println(namesAndReversed3.size) //14
```



iterators.kts

The type of `namesAndReversed3` is also `List<String>` and there are `14` values in it, just like in `namesAndReversed2`.

If you're trying to decide if you should use `map()` or `flatMap()`, here are some tips that will help:

- If the lambda is a one-to-one function—that is, it takes an object or value and returns an object or value—then use `map()` to transform the original collection.
- If the lambda is a one-to-many function—that is, it takes an object or value and returns a collection—then use `map()` to transform the original collection into a collection of collections.
- If the lambda is a one-to-many function, but you want to transform the original collection into a transformed collection of objects or values, then use `flatMap()`.

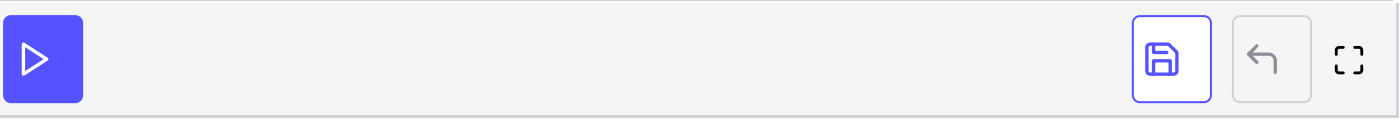
## Sorting #

In addition to iterating over the values in a collection, you can also sort anywhere in the middle of the iteration. You can use as criteria for sorting any details available in that stage of the functional pipeline.

For example, let's get the names of adults from the `people` collection in the sorted order of age, with the youngest `Person's` name first.

```
val namesSortedByAge = people.filter { person -> person.age > 17 }
    .sortedBy { person -> person.age }
```

```
    .map { person -> person.firstName }

println(namesSortedByAge) //[[Paula, Paul, Jill, Jack]
```



iterators.kts

We first filtered the `Persons` who are older than `17` and then used the `sortedBy()` function to sort the `Person` objects based on their `age` property. The collection returned by `sortedBy()` is a new collection, where the elements are in the sorted order of `age`. In the final step of the functional pipeline we used `map()` to extract only the `firstName` properties from the sorted collection. The result is the first names in the order of age from the youngest adult to the eldest adult in the original list.

If we want to sort in the descending order of the age properties, or any other property, we can use the `sortedByDescending()` function. Let's replace the `sortedBy()` call in the above example with `sortedByDescending()` to see this in action:

```
// iterators.kts
.sortedByDescending { person -> person.age }
  //[Jack, Jill, Paul, Paula]
```

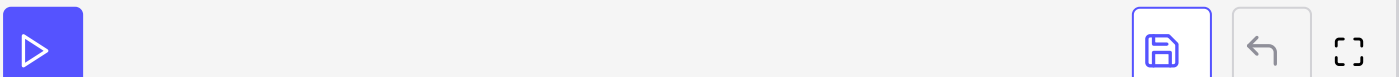
The output of the first names will be in the reverse order, with the eldest `Person's` first name coming first.

## Grouping objects #

The idea of transforming data through the functional pipeline goes far beyond the basics of filter, map, and reduce. You can group or place into buckets objects based on different criteria or properties.

For example, let's group the `Persons` in the `people` collection based on the first letter of their first name, using the `groupBy()` function.

```
val groupBy1stLetter = people.groupBy { person -> person.firstName.first() }

println(groupBy1stLetter)
//{S=[Person(firstName=Sara, age=12), Person(firstName=Sue, age=10)], J=[...
```



iterators.kts

The `groupBy()` function invokes the given lambda for each element in the collection. Based on what the lambda returns it places the element in an appropriate bucket. In this example, `Persons` whose first names start with the same letter are placed in the same bucket or group. The result of the operation is a `Map<L, List<T>>`, where the lambda determines the type of the key of the resulting `Map`. The type of the value is a `List<T>` where `groupBy()` is called on a `Iterable<T>`. In the previous example, the result is of type `Map<String, List<Person>>`.

Instead of grouping the Person, if we want to group only their names, we can do that using an overloaded version of `groupBy()` that takes two arguments. The first parameter is a lambda that maps the element in the original collection to the key. The second lambda maps the element to the value that should be placed into the list. Instead of `List<Person>` for the values, let's create `List<String>` where `String` represents the first names.

```
val namesBy1stLetter =
    people.groupBy({ person -> person.firstName.first() }) {
        person -> person.firstName
    }

println(namesBy1stLetter)
//{S=[Sara, Sue], J=[Jill, Jack], P=[Paula, Paul], M=[Mani]}
```

iterators.kts

Since `groupBy()` is taking two lambdas as parameters, the first is placed inside the parenthesis `()` and the second floats freely outside the parenthesis—see Use Lambda as the Last Parameter.

If instead of grouping the first names, you want to group the age values, replace the second lambda's body with `person.age` instead of `person.firstName`.

With such a variety of powerful tools available combined with elegant and expressive code for internal iterators, you may wonder, Why not use them all the time instead of external iterators or any other alternatives? Appearances can be deceiving and we need to be cognizant of the performance implications of this style.

style.

QUIZ

1 What is the `flatten` function used for?

2 Which functions make up the *functional pipeline*?

3 Which of the following requires a `map()` function to be used?

Retake Quiz

We'll look at that in the next lesson and discuss some alternatives to improve

performance.