Iterate with for

In this lesson, you will be introduced to the for expression.

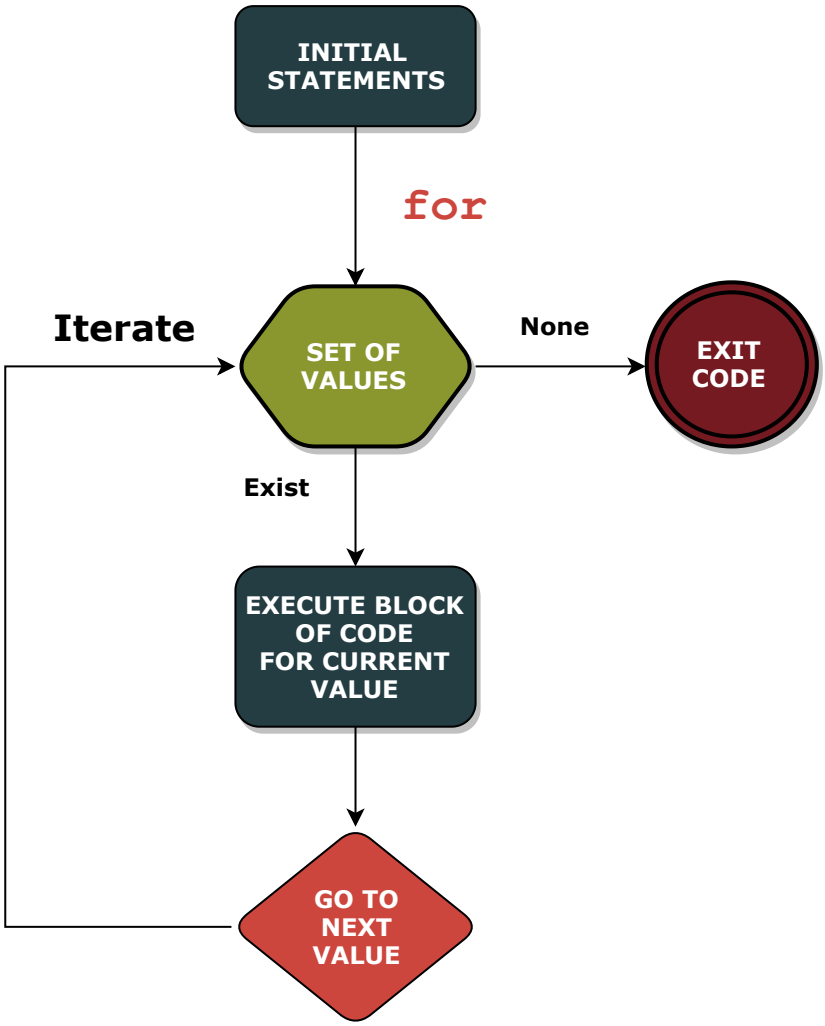
We'll cover the following

Introduction

We sometimes come across scenarios where a block of code needs to be executed multiple times. The `for` expression is one such control structure that does just that. Scala's `for` expression provides a wide variety of ways to express iterations depending on the object you want to iterate over; the simplest being a collection.

Control Flow

Let's look at the general control flow of a `for` expression below.



The for loop iterates over values in a collection. The flow starts by checking if there are any elements in the collection we wish to iterate over. If elements do exist, take the first element and execute the block of code for the current element. Now iterate over all elements in the same manner until you reach the last element of the collection at which point you exit the code.

Syntax

As there are multiple forms of the `for` expression in Scala, there are multiple ways to write them as well. Let's try to generalize the syntax as much as possible.

```
for (generator) {
  block of code
}
```

A **generator** defines a named variable and assigns it to a set of values. It is made

up of three parts: a variable, a `<-`, and a **generator expression**, which is simply a

set of values to be assigned to a variable.

```
(variable <- generator expression)
```

Let's suppose we want to print something five times, our *generator* would take the following form: `(i <- 1 to 5)`. In each iteration, a new value from our set of values (`1 to 5`) will be assigned to the variable `i`. An example will make things clearer.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
for (i <- 1 to 5) {
  println(s"iteration $i")
}
```

The `println` statement is executed five times with `i` being sequentially assigned a new value between **1** and **5** in each iteration. The `for` expression only runs for five iterations as there are five values in our *generator expression*.

Iterating Through a Collection

`for` expressions are ideal for iterating over the elements in a collection when you want to perform the same operation on each element. Let's print the elements of our color array from the previous chapter.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
val colorArray = Array("red", "blue", "yellow")

for (color <- colorArray) {
  println(color)
}
```

The code works the same way as the one we looked at before. However, this time around our set of values are the elements of a collection. Each element, in turn, is assigned to the variable `color` and an operation is performed on it. Let's look at another example where we take the same color array and capitalize the letters.

This code requires the following environment variables to execute:

LANG C.UTF-8

```
val colorArray = Array("red", "blue", "yellow")

for (color <- colorArray) {
  println(color.toUpperCase)
}
```

`for` with `yield`

In all the examples we have looked at so far, the operations performed on the elements are forgotten once the `for` expression is executed. But what if you want to store the values and create new collections?

As discussed in the first lesson, control structures in Scala have return values. To obtain the return value of a `for` expression, we can use the `yield` expression which does exactly what its name implies: *yield* the return value.

```
val variableName =
  for (generator)
    yield body
```

`yield` must be placed before the entire *body* or block of code.

Let's take the previous example where we are capitalizing the elements of an array. But this time we will create a new array of the capitalized elements.

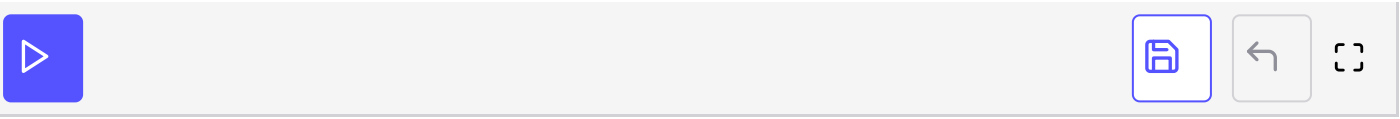
This code requires the following environment variables to execute:

LANG C.UTF-8

```
val colorArray = Array("red", "blue", "yellow")

val newArray =
  for(color <- colorArray)
    yield color.toUpperCase

// Driver Code
for (newColor <- newArray)
  println(newColor)
```



In **line 3** of the above code, we are storing the value of `color.toUpperCase` in a new array.

Filtering with `if`

Sometimes, we don't want to operate on each element of a collection, rather only on specific elements. Adding an `if` expression to the `for` generator allows us to filter out the elements we aren't interested in.

Let's look at an example where we start with an array of integers from **1** to **10**. We want to create a new array of even integers using the initial array. How would we go about doing that?

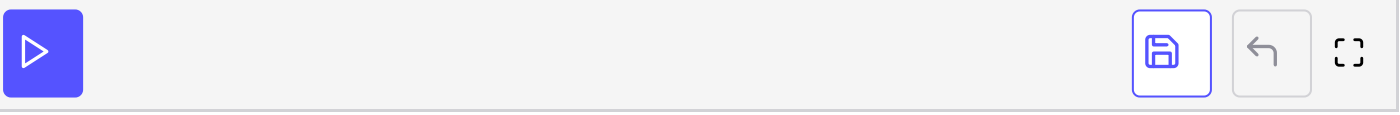
This code requires the following environment variables to execute:

LANG	C.UTF-8

```
val intArray = Array(1,2,3,4,5,6,7,8,9,10)

val evenArray =
  for (element <- intArray if element % 2 == 0)
    yield element

// Driver Code
for(i <- evenArray)
  println(i)
```



In **line 4**, of the code above, we have added a condition to our generator: `element % 2 == 0`. The `for` expression will only iterate over the elements of the collection for which this condition is `true`, i.e., even numbers.

In the next lesson, we have a challenge for you to solve for yourself.