For Loops

Let's take a look at the key features of for loops.

WE'LL COVER THE FOLLOWING

The age-old `for` loop has become a staple in the world of programming. It allows us to specify a range of numbers over which we want our loop to run.

It also has another component called the **iterator**. The iterator is responsible for keeping track of the iterations. Initially, its value is the start of the loop range, which changes with each iteration.

The return type of a `for` loop is expected by the compiler to be `unit`. Hence, it is not a good practice to return things from the `for` loop to the program outside its scope.

The Structure

Below, we can find the basic template of a `for` loop.

```
for (iterator in start to end) {
  set of operations
};
```

The `start` and `end` identifiers specify the range of the `iterator`.

There is a common universal practice to call the iterator `i`, but it could also be anything we find appropriate.

The Syntax

Here's a simple `for` loop in action:

```
for (i in 1 to 5) {
  Js.log(i);
}
```

As we can see, the starting and ending values of the loop are inclusive in the iteration. So, a loop from `1` to `5` would have 5 iterations, whereas, a loop from `0` to `5` would have `6` iterations.

Another important observation is that the value of our iterator, `i`, is available throughout the loop. We'll soon see why this is useful.

The iterator has an increment of `1` in each iteration until the end of the list is reached.

Traversing an Array

This is one of the most common uses of loops. The value of the iterator can be used as the index for an array as long as it remains less than the array's maximum index.

Let's traverse a simple array through a `for` loop:

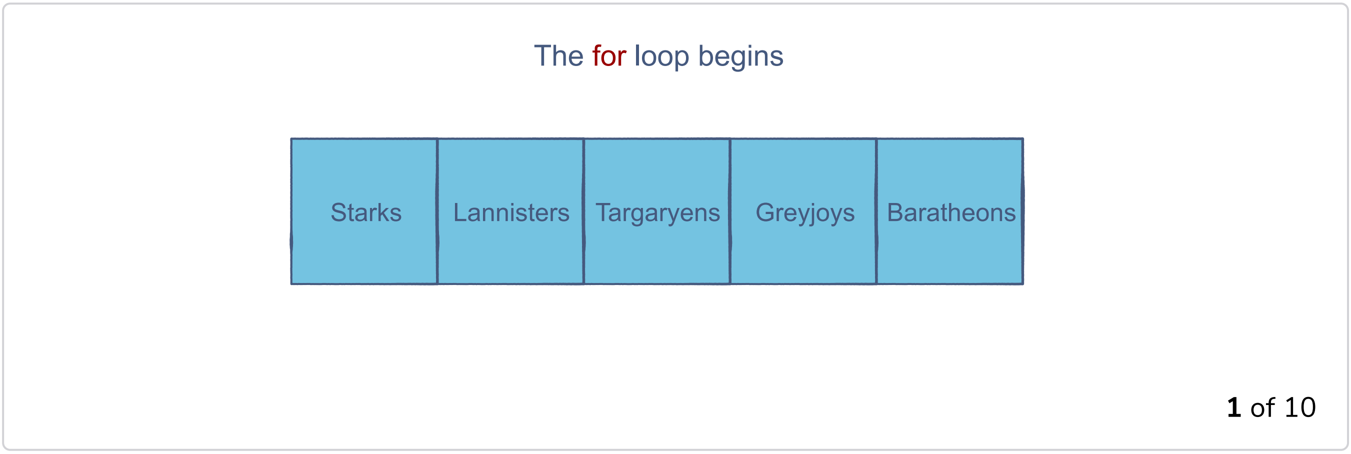
```
let arr = [| "Starks", "Lannisters", "Targaryens", "Greyjoys", "Baratheons" |];

let max = Array.length(arr) - 1;
for (i in 0 to (max)) {
  Js.log(arr[i]);
};
```

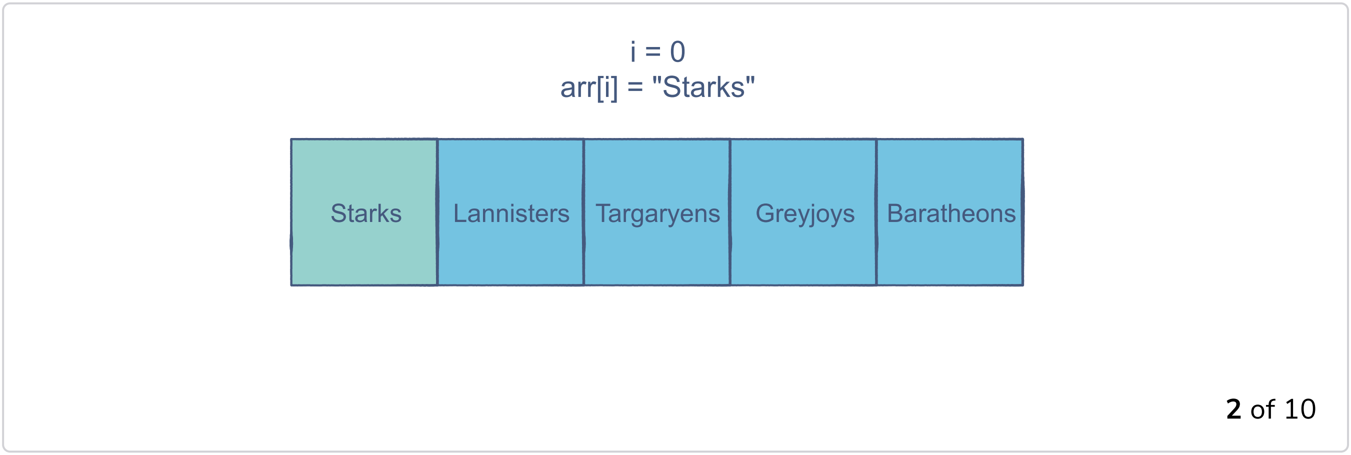
We can perform all sorts of operations within the loop.

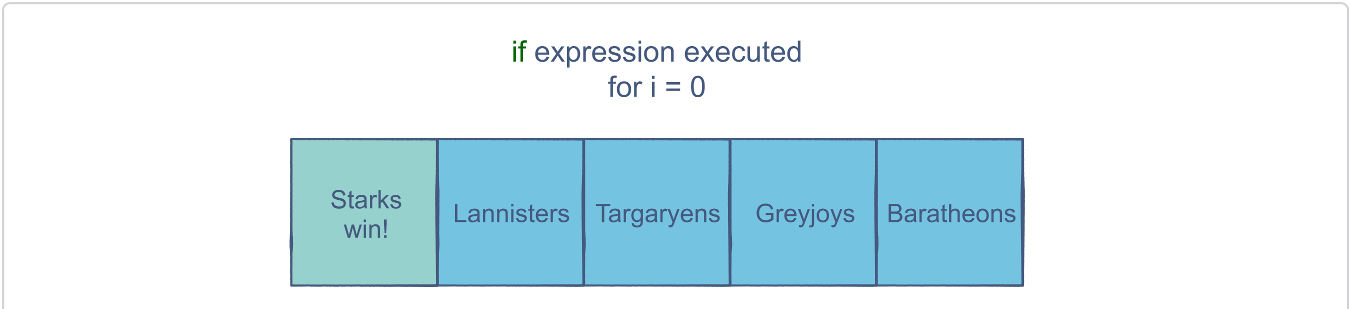
Here's an example of the array above being manipulated through functions and loops.

```
let arr = [| "Starks", "Lannisters", "Targaryens", "Greyjoys", "Baratheons" |];
Js.log(arr);

let win = (string) => string ++ " win!"

let lose = (string) => string ++ " lose..."

let max = Array.length(arr) - 1;
for (i in 0 to (max)) {
  if (arr[i] == "Starks"){
    arr[i] = win(arr[i]);
  }
  else {
    arr[i] = lose(arr[i]);
  }
};
Js.log(arr);
```

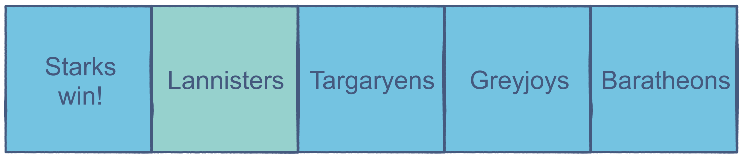
The following illustration takes us through the whole process step-by-step:



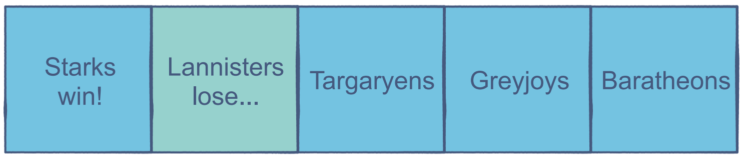




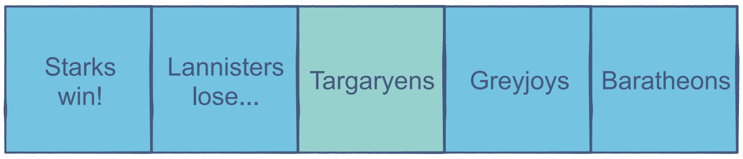
i = 1



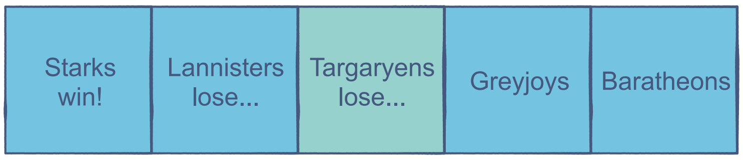
else expression
executed for i = 1

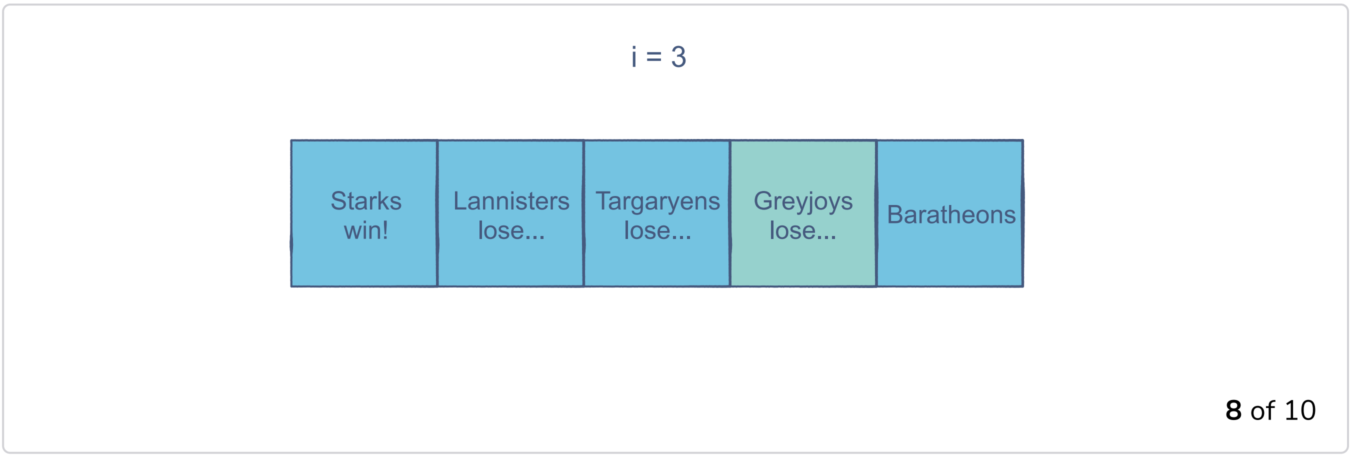


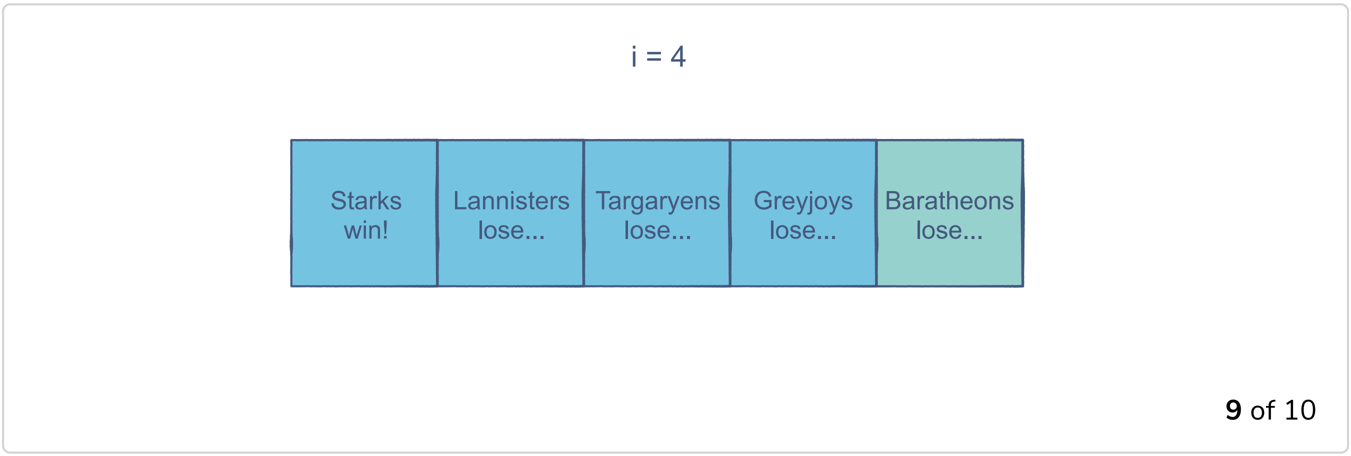
i = 2

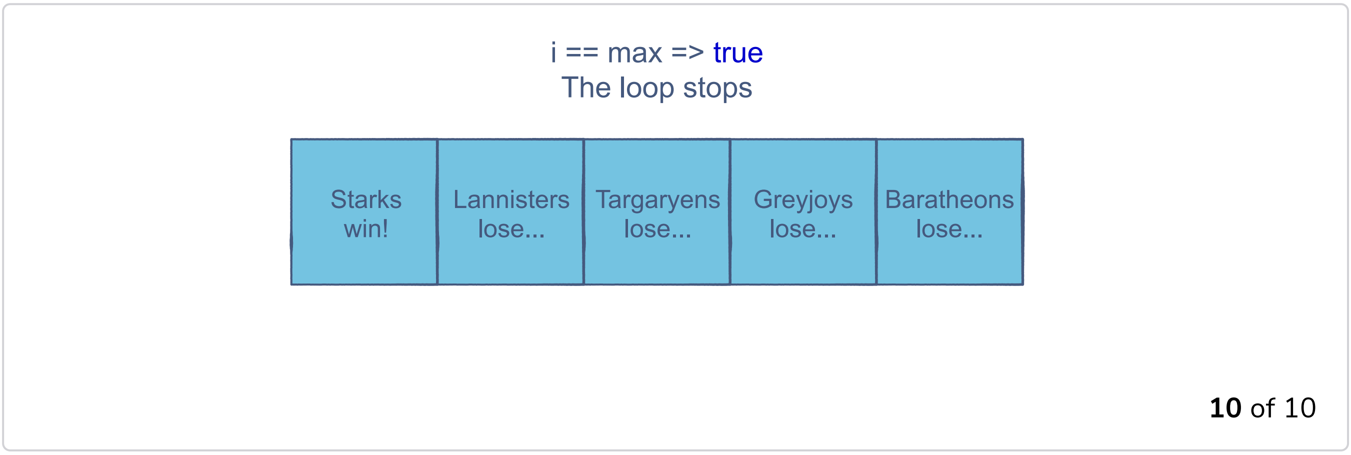


The process repeats









Iterating Backwards

The `for` loop structure also allows us to iterate in reverse, i.e, the value of the iterator decreases in each iteration.

To use this feature, we must replace the `to` keyword with `downto`:

```
for (i in 5 downto 0) {
  Js.log(i);
};
```

Next, we'll study the `while` loop.