Recursive Functions

In this lesson, you will be given a brief introduction to recursion and go over how recursion is implemented in Scala.

We'll cover the following

Recursive functions play an essential role in functional programming. But what are *recursive functions*?

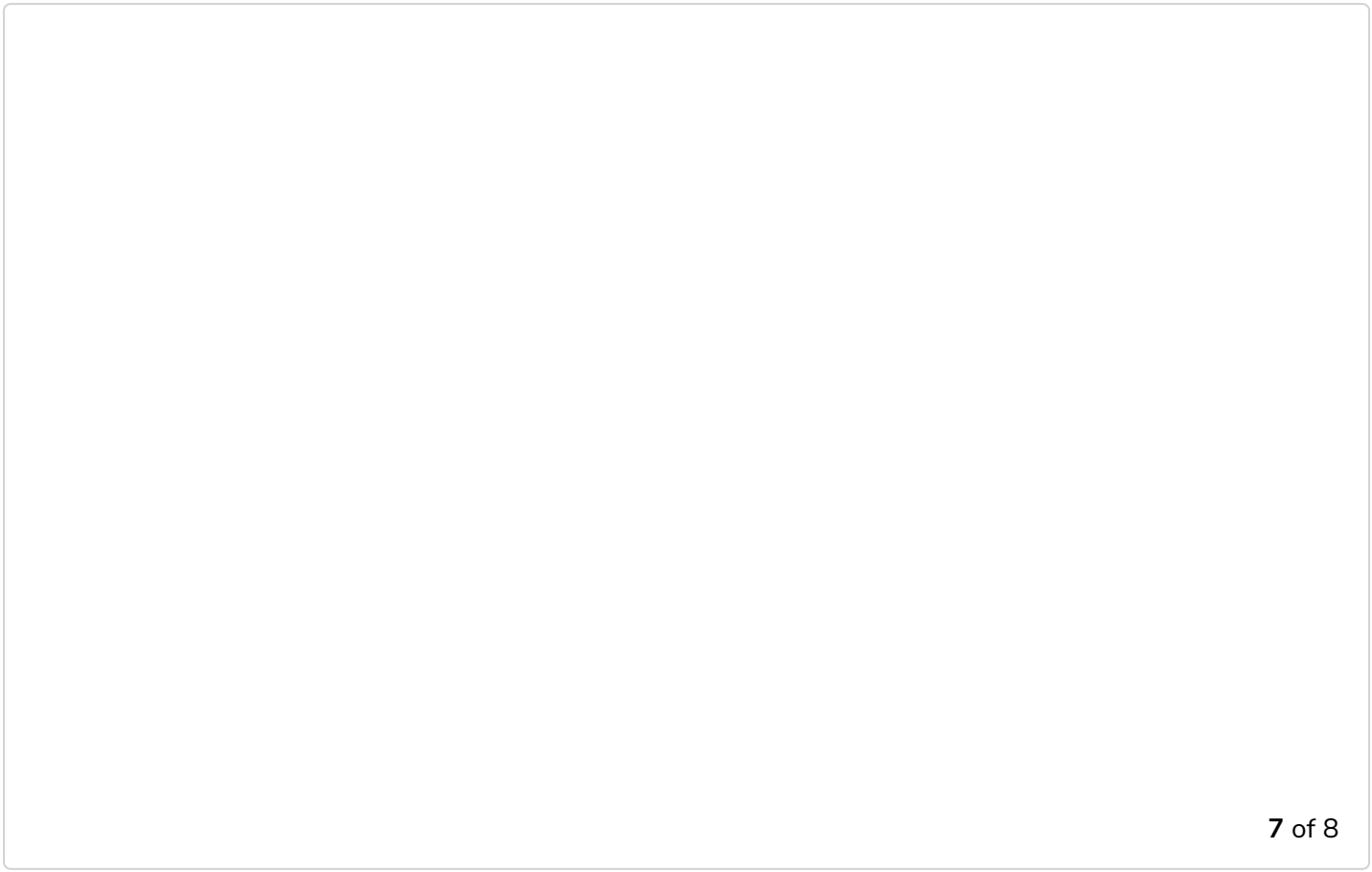
Recursive functions are functions which call themselves in their own function body. This may seem a bit strange right now, but let's see how this works.

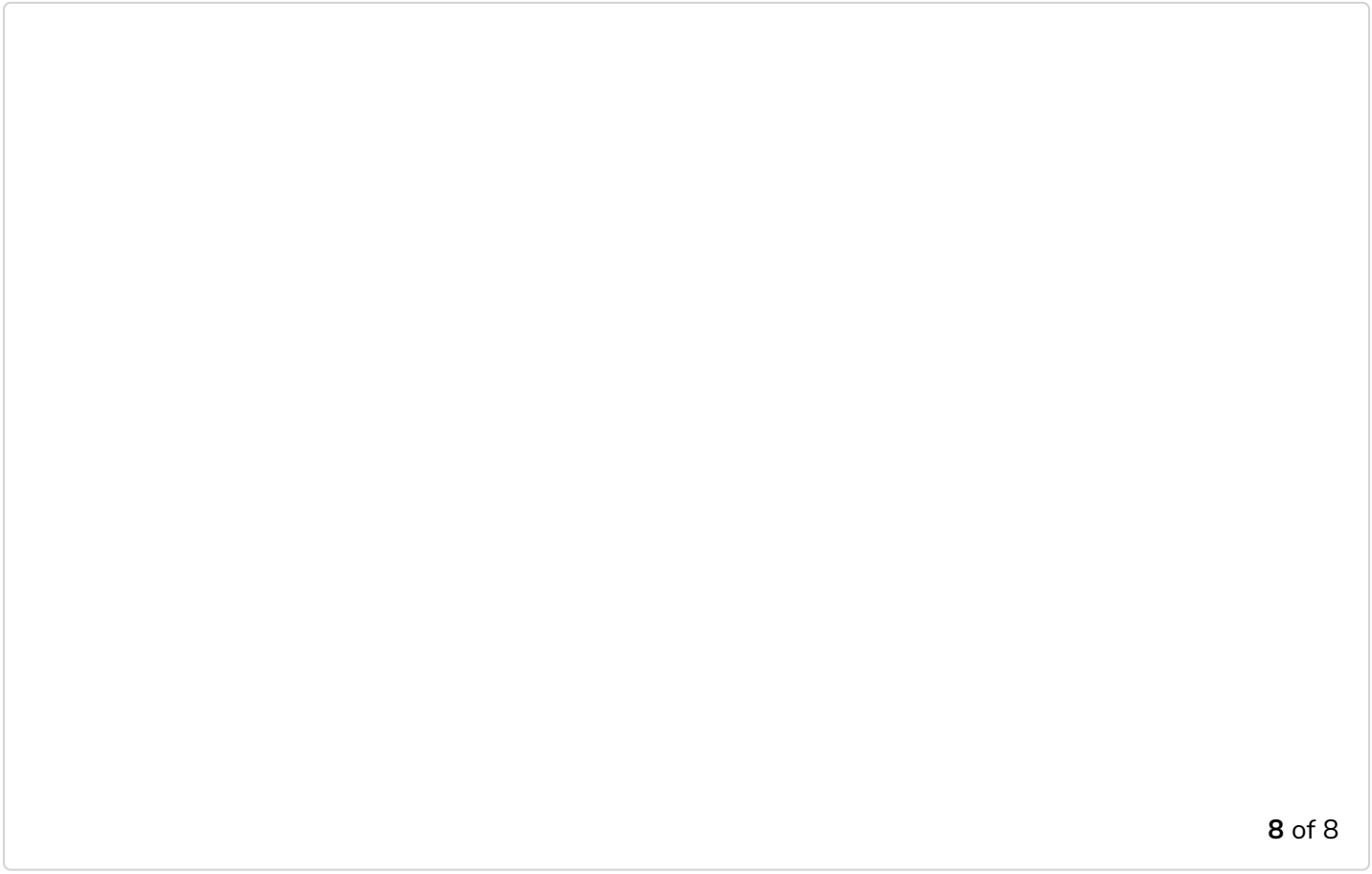
Recursion

Recursion is the process of breaking down an expression into smaller and smaller expressions until you're able to use the same algorithm to solve each expression.

A recursive function is made up of an `if-else` expression. The `if` represents the *base case* which is the smallest possible expression on which an algorithm will run and the `else` represents the *recursive call*; when a function calls itself, it is known as a recursive call. The recursive function will keep calling itself in a **nested** manner without terminating the call until it is equivalent to the base case in which case the algorithm will be applied, and all the function calls will move in an outward manner, terminating before moving on to the next one, reducing themselves until they reach the original function call.

Let's look at recursive calls as boxes within boxes.





Scala's Implementation

The implementation of factorials is a very famous recursive problem. A factorial of a number is achieved by multiplying all consecutive numbers starting from **1** till the number in question. So, the factorial of **4** (represented as **4!**) is equivalent to **1** x **2** x **3** x **4** = **24**.

Let's look at how we would implement this in Scala.

This code requires the following environment variables to execute:

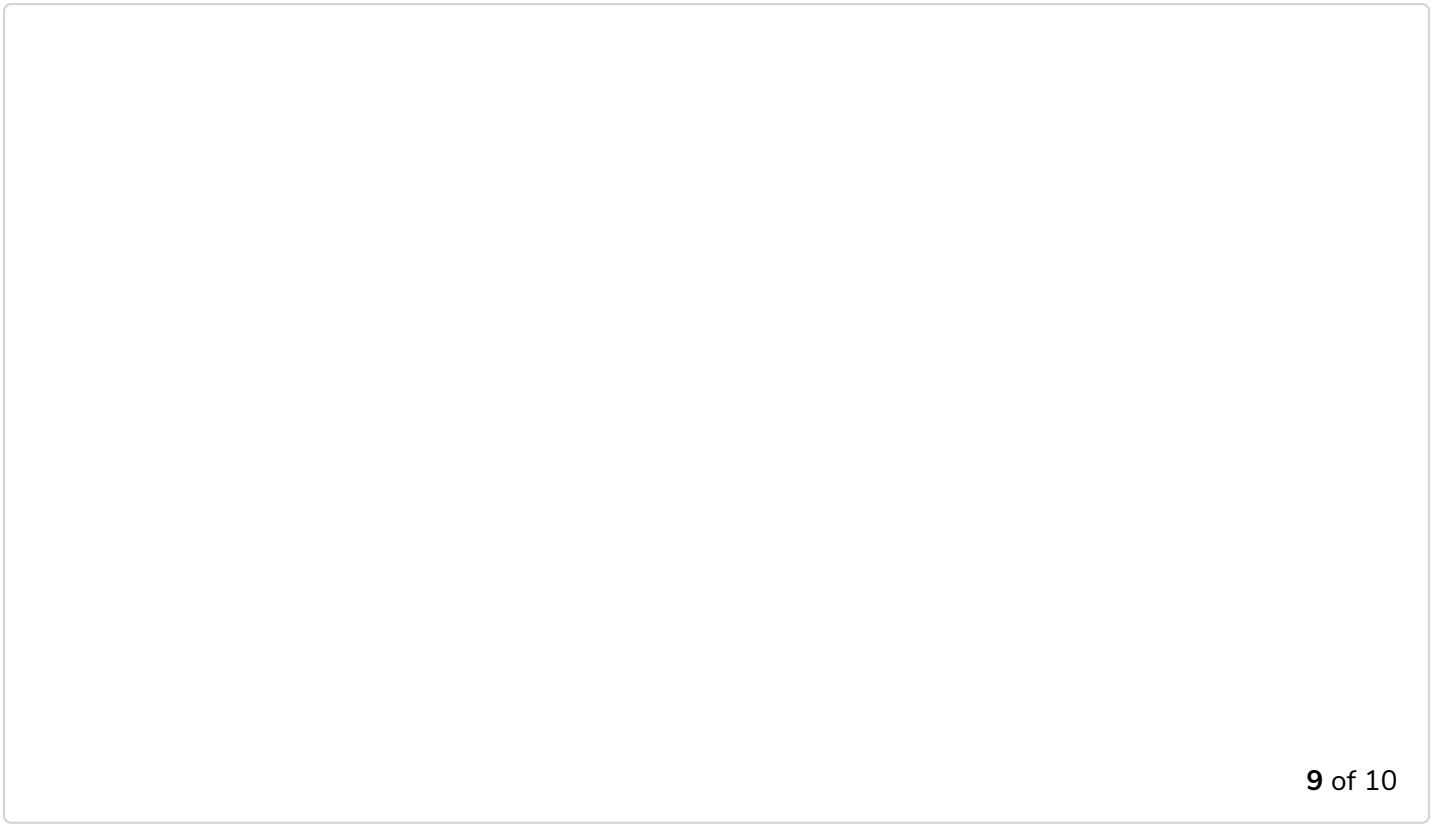
LANG	C.UTF-8

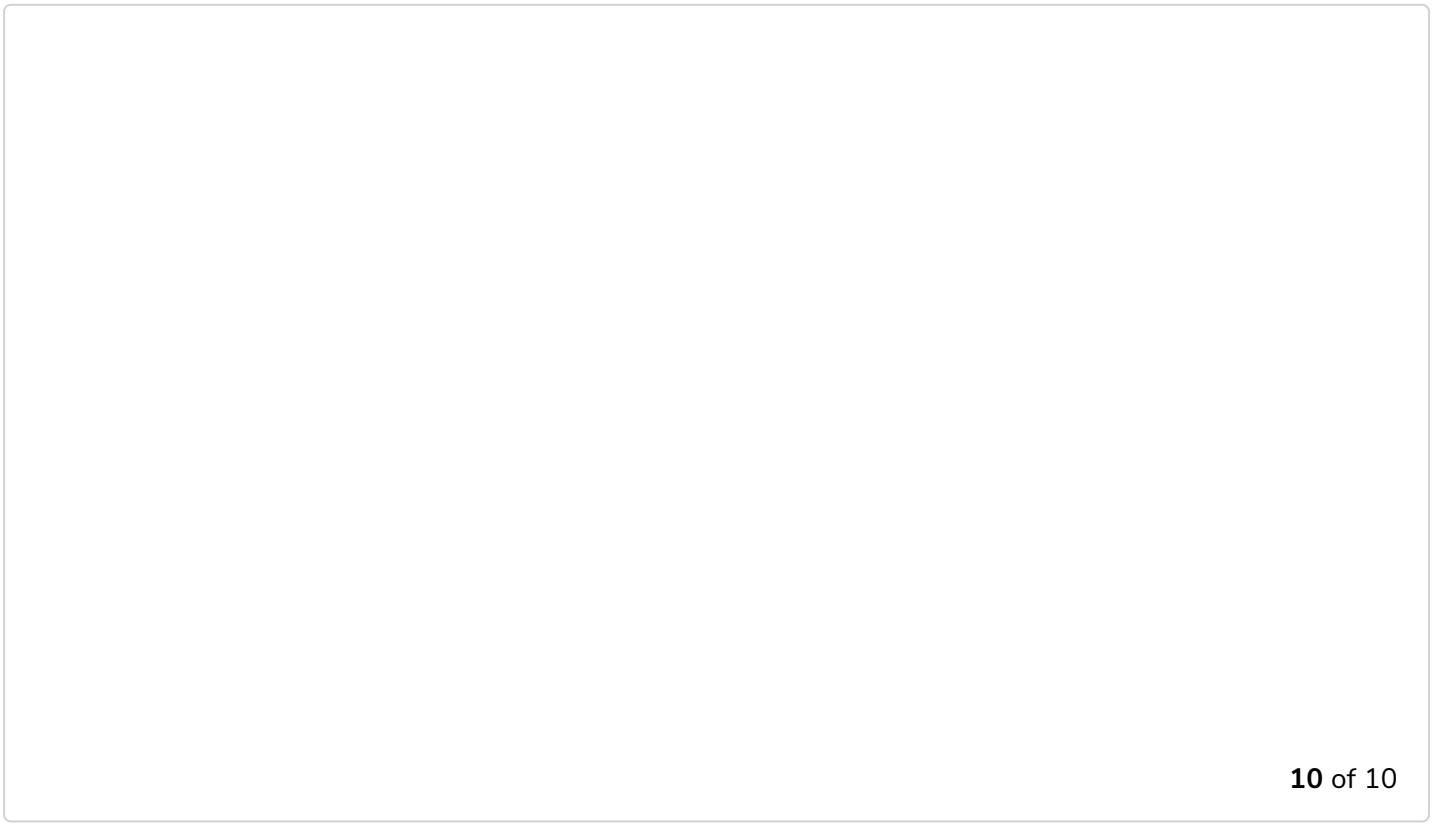
```
def factorial(x: Int) : Int = {
  if(x == 1)
    1
  else
    x * factorial(x-1)
}

// Driver Code
print(factorial(4))
```

> Unlike a basic function, recursive functions require you to specify the type of the return value.

`factorial` is a recursive function which takes one parameter; the number whose factorial is to be calculated. The base case is if the number is **1** in which case all we have to do is return **1** as that is its own factorial. We cannot break down the expression farther than this, where the factorial of a number is the number itself.

However, if the number is not equal to **1**, the `else` expression will be executed in which the return value is the number being multiplied with the factorial of the number before it. This will continue until we pass **1** in our recursive call. Let's look at a graphical implementation of how this works.





Do you see how the result of an inner function call is used to terminate its outer function call?

Implementing Recursive Functions Using `match`

As discussed above, recursion follows an `if-else` pattern. We can also implement

recursive functions using `match`. The basic concept is exactly the same as before.

There is a base case which is handled by the **first case**. The **second case** handles the recursive call. It is exactly like `if-else` with the **first case** representing `if` and the **second case** representing `else`.

This code requires the following environment variables to execute:

LANG	C.UTF-8

```
def factorial(x: Int) : Int =  x match{
  case 1 => 1
  case x => x * factorial(x-1)
}

// Driver Code
print(factorial(4))
```

Recursion is a difficult concept to wrap your head around, but once you do, it is an extremely powerful tool you can use in a functional programming language.

In the next lesson, you will be challenged to write your own recursive function.