Scala fundamentals - Lazy values and Evaluation

Aims

We will be covering the following topics:

- Lazy vals and evaluation

Prerequisites

You have an understanding of `vals` and `defs` in Scala.

Lazy vals and evaluation

When we use the `val` keyword to define a constant value, it evalutes at the time of definition. From the point of evaluation, it will always have the same type and the same value. It will never be re-evaluated. This is known as **eager evaluation**.

```
val x = "Hello"
print(x) // Hello
x = "Goodbye"
11: error: reassignment to val
       x = "Goodbye"
```

When we use the `def` keyword to define a function, the body of the function is evaluated only when we make a call to it.

```
def sayHello(name: String) = print(s"Hello $name")

sayHello("jenny") // Hello jenny
sayHello("adam") // Hello adam
```

This type of evaluation - which happens at the time we call the function - is known as **lazy evaluation**. If we don't make a call to `sayHello`, it will never be evaluated. Every time we call it, it will be re-evaluated.

In some situations, we might want to have a `val` that behaves in a similar way: only evaluating when we call it. This is possible through the use of the `lazy` keyword to define a `lazy val`. By doing this, we define an immutable `val` that will still be only evaluated once, but this evaluation will not occur until the `val` is used.

Tip

> It is not possible to make a `lazy var`.

> `lazy val` and `var` are different in that `var` is eagerly evaluated - at the time of definition - while `lazy val` is, as it's name suggests, lazily evaluated.

Code example of different evaluation types

The following Scala program will print foo, then bar then 10.

```
val x = { print ("foo") ; 10 }
print ("bar")
print (x)
```

If we make `x` into a `lazy val`, the following Scala program will instead print bar, then foo, then 10, since it delays the computation of x until it's actually needed.

```
lazy val x = { print ("foo") ; 10 }
print ("bar")
print (x)
```

Further reading

- Scala: The Cost of Laziness

Task

- Implement the above code in a Scala worksheet and try it for yourself
- Can you think of any code you have implemented recently that would benefit being lazy?

When to use lazy evaluation

There will be times when you want to lazily evaluate a value. For instance, you may have a value that makes HTTP calls to another service to calculate a result, this may be an expensive operation to perform as soon as your application starts. For memory efficiency, you may only want to instantiate the value once you're absolutely sure you're going to use it, therefore, you would make the value lazy i.e. `lazy val`. By choosing to use a value you will only evaluate the value of this once and only once. Meaning you won't have to determine the result each time it is used.

Alternatively, you may decide to use a `def` with no arguments for instance to enforce that the value is lazy, and, the value has to be calculated/evaluated each and every time it is used. This would be useful for HTTP requests for instance, or creating expensive objects as the properties of the objects may change based on how the user has interacted with your application.

Recap

Here we have discussed how the order of evaluation changes when using a `val`, `lazy val` and `def`. We've also talked about when to make a value lazy or change the order of evaluation, this is useful when not wanting to instantiate expensive objects, or perform a long computation.

Resources

- Scala: The Cost of Laziness