Lecture 21: Scala Mutation, Loops, and Exceptions

11:00 AM, Mar 14, 2021

Contents

Objectives

By the end of these notes, you will know:

- The relationship between var, val, and private modifiers
- Mutable vs immutable data structures in Scala
- How to write loops in Scala
- How to work with exceptions in Scala

1 Var, Val, Public, Private

In Java, we controlled read and write access by marking fields private. How do var and val relate to private?

Var and val are more lightweight. Private limited all access to a field, forcing you to write getters and setters when necessary. There was no way to give only read access in Java unless you made a field private and provided only a getter. Val lets you do that with less code.

But private had another angle, which was that it prevented subclasses from accessing a field (which is sometimes quite useful). Val and var say nothing about this concept. If you want that control, you need .... private (or protected) in Scala.

Yes, Scala also has private and protected (public is on by default). What do they mean? Let's look at our `Listing` class again, this time with the `rate` field marked as private.

```
class Listing(val name : String, private var rate : Double, val sleeps : Int
    ) {
  var amenities = List("hairdryer")

   override def equals(that: Any): Boolean = that match {
    case that: Listing => that.name == this.name && that.sleeps == this.
        sleeps
    case _ => false
  }
}
```

Var and val control what all classes can do to a field (the class containing the field as well as outside classes). Private and protected control which classes can perform var and val actions on a field. A private var can only be read or written by the class containing the field (not any outside classes or

subclasses). A protected var extends those privileges to subclasses. A private val can be read by the class containing the field, but by no other class.

Upshot: private is orthogonal to var/val, even though both can restrict some permissions of outside classes.

1.1 Private and equals

One other tidbit here: if we mark a field as private, how can we use it in writing an equals method? Wouldn't equals need to see inside the `that` object to look at its private field?

Yes, and this brings up a nuance. Private fields are readable by *other objects of the same class*. Otherwise, you wouldn't be able to write an `equals` method. This is also true of Java.

2 Val and Var on Data Structures

Our `Listing` example annotated numeric fields with var and val. What happens if the type of our field is a data structure, such as a list or hashmap? How do var and val limit what we can do with those data structures? And then how does mutability interact with this? We've already seen that lists are by default immutable in Scala (though there is a mutable version as well).

We'll use these questions as a chance to learn the hashmap notation in Scala as well. Let's make a hashmap with String keys and numeric values.

```
// create a hashmap with keys ``A'', ``B'', and ``C''
map1 = Map(``A'' -> 1, ``B'' -> 2, ``C'' -> 3)
// get the value of key ``A''
map1(``A'')
// extend a hashmap with a new key-value pair
map1 + (``D'' -> 4)
```

We could annotate `map1` as val or var. What difference would it make?

To think through this, consider the following picture of the environment and memory heap:

If we say "map1 is mutable", what might we be saying? What's a candidate to change in this picture? We could be talking about any of the following things:

- Can the set of keys change?
- Can the values associated with existing keys change?
- Can `map1` be set to refer to a different hashmap?

Our question is, how do the answers to these change depending on whether (a) the hashmap is mutable or (b) marked as one of val or var.

The choice of val vs var controls the arrow from the environment to the heap. If you define `map1` as a val, you may not associate the name `map1` with a different hashtable within memory. This is the same rule as applied for numeric fields (you can't write `field = ...`.

Mutable vs immutable controls changes to the hashmap within the heap. If you create an immutable hashmap (as the example above does), you may not add keys or update values within the hashmap.

What, then would the following code do? Would it compile?

```
map1 = Map(``A'' -> 1, ``B'' -> 2, ``C'' -> 3)
map1 + (``D'' -> 4)
```

Sure, it compiles fine. The second line simply creates and returns a new hashmap, with the same key-value pairs as `map1`, as well as the additional pair ``D'' -> 4. This is the same as for immutable lists: you can always write `4 :: myList`, and the value of `myList` remains unchanged.

What if we want to be able to mutate the contents? How should we have defined `map1`?

```
val map2 = scala.collection.mutable.Map(``A'' -> 1, ``B'' -> 2, ``C'' -> 3)
```

Now, you could write something like

```
map2(``A'') = 5
```

and the value for "A" within `map2` would now show as 5.

3 Var/Val Takeaways

The main takeaway here is that when you set up a field whose value is a data structure, you have two choices to make:

1. Can the inner contents be changed? If yes, make it mutable.
2. Can the field be assigned to a different data structure in memory? If yes, mark it a var.

In general, use the most restrictive option that works for your application. If you don't need to change the contents (and don't want anyone else to change them either), make your data structure immutable.

This same guideline holds for any other language you work in. Even if those languages don't give you convenient keywords for mutability, you can make fields private and control what the outside world can do with your data. Allow as little update as possible unless there's a reason why updates are necessary for your application.

4 Scala Exceptions and Other Tidbits

The code handout for today's lecture (linked to the lectures page) shows several methods that one might write to manage hotel reservations. The code features:

- An array of Optional strings (to capture reservations on the days of a 30-day month), along with code showing how to set default values in Scala Arrays
- Two versions of a method to count how many days are unbooked in the month
- Two versions of a method to print out the reservations in the Array
- A method to reserve rooms, after checking that the requested dates are available. The checking method throws an exception if some day is booked
- An alternative method for checking whether rooms are available, this time returning a Boolean indicating whether the room is free.

These examples brought up several bits of useful information about Scala:

- *Option* types are supported in many languages as an alternative to using null to denote missing data. Option types have two kinds of values: `None` (for no value) and `Some(value)`. Option types are considered better than null because they signal an intentional lack of value, rather than an error (such as from an uninitialized data structure).
- The `flatten` function (which does not need to be called with parens since it takes no arguments) takes a list of lists or arrays and concatenates them into a single list. It also removes any None options, leaving only the Some elements in the resulting concatenated list.
- `foreach` is like `map`, except it doesn't save the results of the function on the list inputs. It is suitable for situations that need to update memory or print something for each item in a list.
- `forall` on a list or array takes a function on list elements that returns boolean; it returns true if that function returned true on all list elements.
- You create new exception classes the same way as you did in Java.
- You throw exceptions similarly to in Java, but you don't annotate methods with `throws` annotations.
- Exception `catch` blocks are written uses the `case` notation, which differs from the notation in Java. Conceptually, however, exceptions behave the same way in both Scala and Java.

Mostly, this code handout is provided as a reference so you can see how to set up certain operations in Scala.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.