Lab 8: Introduction to Scala
12:00 PM, Mar 11, 2020

Contents

1 Getting Started 1

2 Val and Var on Data Structures 1

3 Suspect Name Tracking 3
   3.1 Vals vs Vars 3
   3.2 Public vs Private 4

4 Lunch Time! 5

Objectives

By the end of this lab, you will know:

- mutable and immutable data types in Scala
- Scala traits

By the end of this lab, you will be able to:

- use vals and vars, as well as mutable and immutable data structures
- use and combine traits effectively in Scala

1 Getting Started

First, copy over our source files:

cp /course/cs0180/sol/lab08/sol/* ~/course/cs0180/workspace/scalaproject/sol/lab08/sol

2 Val and Var on Data Structures

When you, the illustrious CIT Detective, get back to your office after your lunchbreak at Geoff’s, you realize that someone has been changing the notes you keep on the computer to track potential suspects. A file has been added consisting of the text “I hacked your computer. Your files are all mine now. You got hacked.” Confusingly, it doesn’t look like anything important was changed or deleted, but you don’t want to let it happen again. To prevent this attack, you need to change the mutability and accessibility of the notes. Val
and Var, public and private, immutable and mutable, what do these keywords even mean? You need to turn to cs0180 for help.

Our Listing example from lecture 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? How does mutability play into this (we’ve already seen that lists are immutable by default 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.

```scala
// 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 could be changed 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? Write down your answers.

```scala
map1 = Map(''A'' -> 1, ''B'' -> 2, ''C'' -> 3)
map1 + (''D'' -> 4)
```
What if we want to be able to mutate the contents?

Say you want to be able to write something like

\[
\text{map2(''A'') } = 5
\]

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

How should we have defined map1?

### 3 Suspect Name Tracking

Now, you need to build a way to store the names of your suspects such that no future hackers could change your notes. You also want a way to clear a suspect’s name, in case you later discover evidence they didn’t commit the crime and want to strike them from the notes.

**Task:** Go ahead and open our source file, `Names.scala`. You will be working with this file for this section.

**Take 1: Val Array**

You decide to first try putting names in an array:

```scala
Array("Jefferson", "Justin")
```

Recall that arrays are mutable. You want to make a `Names` class whose constructor takes in a private `val` array of suspects. Here is the class declaration:

```scala
class Names(nameList: Array[String])
```

**Task:** Fill in the class `Names`, which contains the method `clear(name: String)`. This method should change all instances of `name` in the array to “CLEARED”.

**Hint:** Even though your array is stored as a `val`, that only means the array itself (i.e. its size and type) cannot change, but the contents of the array can– arrays are mutable data structures!

**Note:** We did not use the keywords `private` or `val` in this class declaration, because that is the default for Scala!

**Take 2: Var Immutable List**

After looking at the suspect list, you figure that he might want to add more suspects to the list in the future. You realize that an immutable `List` might be better suited for this task than an array!

**Task:** Fill in the class `Names2` which takes in a `List` of names as a `var` and includes a `clear` method that works with the list of names.

**Hint:** Try using `map` (the higher order procedure)!

**Take 3: Val Immutable List**

You realize that you don’t want just anybody modifying the list of suspects. Make one more class, `Names3` which takes in just a `List` of names as a `val`.

**Task:** Since the `nameList` is now completely immutable, you cannot modify its contents, nor can you reassign it to a modified list. You need to write `clear` to return a new instance of `Names3` with the correct internal list. Now the data is completely secure!

**Task:** Why do we now need to return a new instance of `Names3`, instead of having `clear` return `void` as it did in the other two implementations? Write this down on a piece of paper.

**Task:** Use `NamesTest.scala` to test your code:
Warning: Although you could technically write Scala programs with only vars and no vals, you should avoid this. Not only will you be penalized in CS 18 if you use var when you could use val instead, but val increases the clarity of your code, because you don’t need to worry about all those variables’ values changing when you don’t want them to.

3.1 Vals vs Vars

Should you declare your collection as a val or a var? Depends on what you’re looking for! Fill in the following table detailing when (or if) you use each keyword with mutable and immutable collections:

<table>
<thead>
<tr>
<th></th>
<th>Immutable Collection</th>
<th>Mutable Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Val</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.2 Public vs Private

You also realize that you don’t want just anybody to be able to access the list of names, even if they wouldn’t be able to edit them. How can you restrict other classes from accessing this field?

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.

```scala
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.

Add the private keyword to the names List and convince yourself that it does in fact behave as described above. Then, for the following combinations of val/var and public/private, describe an example where you think that that compination of keywords is needed:

- public var
- private var
- public val
- private val

| You’ve reached a checkpoint! Please call over a lab TA to review your work.

## 4 Lunch Time!

After finishing your suspect finder, you develop a bit of an appetite, and decide to order yourself a delicious sandwich from your favorite restaurant, Geoff’s, over Snackpass. However, when you arrive, you find that they have no record of your Snackpass order ever occurring; the only recent Snackpass order is for Thien, the CS18 TA, who ordered the “David Cicilline”, the exact same sandwich you ordered! You wait around, and when Thien comes to pick up his sandwich, you confront him. Knowing of your investigative prowess, he caves immediately, admitting that he’s been hacking Snackpass to steal other people’s orders because he doesn’t know how to make himself lunch. Taking pity on the poor soul, you decide to teach him how to make a sandwich.

**Task:** In a new Scala file, define a class `Lunch` with a single field `ingredients`, represented as a list of Strings.

**Task:** Within the `Lunch` class, define the `makeSandwich` method, which just returns `ingredients`.

Now, you can make some delicious, albeit simple, sandwiches for lunch, like this:

```scala
scala> val dessert = new Lunch(List("Marshmallows", "Nutella", "Apples"))
scala> dessert.makeSandwich
List("Marshmallows", "Nutella", "Apples")
```

Next, you will define a few traits that you will mix in to your `Lunch` class to augment the functionality of `makeSandwich`.

**Task:** Write the following traits, each of which extends `Lunch` and overrides the `makeSandwich` method as described:

- `PeanutButter`: returns all the ingredients, with "Peanut Butter" prepended.
- `Bread`: returns the ingredients with "Bread" added to both ends.
- `HoldTheAnchovies`: returns the ingredients, without any "Anchovies".
• DoubleIngredients: doubles every ingredient, such that each appears twice in a row.

**Hint:** You can (and should!) define all these traits in the same Lunch.scala file!

**Hint:** To help get you started, if one of our requirements was TinySandwich, which removes the first half of the ingredients provided, we would have:

```scala
trait TinySandwich extends Lunch {
  override def makeSandwich = super.makeSandwich.drop(super.makeSandwich.length / 2)
}
```

**Hint:** As always, you should strive to produce elegant and concise programs. To that end, consider using higher-order functions to implement some of these traits, if appropriate.

By mixing in these traits, you can make still more delicious sandwiches, like this one:

```scala
scala> val lunch = new Lunch(List("Jelly")) with PeanutButter with Bread
scala> lunch.makeSandwich
List("Bread", "Peanut Butter", "Jelly", "Bread")
```

Although we use traits to model interfaces in Scala, traits are richer than Java interfaces. In Java, a class can implement multiple interfaces, and no conflict arises if two of those interfaces declare identical methods with identical signatures because interfaces do not include implementations.

In Scala, however, traits do (generally) include implementations. Scala resolves any conflicting method definitions by “linearizing” them: the rightmost mixin receives the initial method call. Moreover, each time a trait is mixed in, it overrides any existing methods with conflicting type signatures.

**Hint:** All your traits should override `makeSandwich`. To mix in more than one of them, your `makeSandwich` methods should call `super.makeSandwich`.

**Hint:** Here’s how you can fold in Scala:

```scala
myList.foldRight(List[String]())((curr, rest) => /*something with curr and rest*/))
```

On that line, the `(List[String]())` is the base case for fold, and `curr` is referring to the current element you’re folding over, whereas `rest` is the result of the rest of the fold!

**Task:** Now it’s your turn to make some sandwiches! Here’s how. First, in the same Lunch.scala file, create a Lunch object, containing a `val List("Anchovies", "Cream Cheese", "Pickles")`. Then, create new Lunches that take in this list, mixing in various sequences of the traits you just defined, to create the following delicious sandwiches:

- List("Bread", "Anchovies", "Cream Cheese", "Pickles", "Bread")
- List("Peanut Butter", "Cream Cheese", "Pickles")
Note: Put these sandwiches in a companion object to your Lunch class. Name the sandwiches monday, tuesday, ... friday, respectively, so that a call to monday.makeSandwich returns List("Bread", "Anchovies", "Cream Cheese", "Pickles", "Bread"), and so on.

Here’s an example to get you started:

```scala
object Lunch {
  val myList = List("Anchovies", "Cream Cheese", "Pickles")
  val monday = new Lunch(myList) with ???
}
```

Task: Uncomment our LunchTest file and use it to test your implementation with the days of the week. Then, write at least 2 more sandwiches below the weekday Lunches (and corresponding tests)!

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

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