Lab 8: Introduction to Scala
12:00 PM, Mar 14, 2018

Contents

1 Getting Started 1

2 Vals vs Vars 1

3 Team Rocket’s Shenanigans 2

4 Lunch Time! 3

5 Just for Fun: Currying Foods (pun intended) 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/src/lab08/src/* ~/course/cs0180/workspace/scalaproject/src/lab08/src
cp /course/cs0180/sol/lab08/sol/* ~/course/cs0180/workspace/scalaproject/sol/lab08/sol

2 Vals vs Vars

There are two keywords for declaring variables in Scala: val and var. Val is used for immutable data—the kind we studied in CS111, CS 17 and CS 19.

The latter is used for mutable data—the kind we have focused on in CS 18. That sounds simple enough: use val when declaring a variable whose value will never be reassigned, and use var otherwise.
Scala also has immutable and mutable collections (i.e. Maps, Sets, Lists). You can tell which type of collection you are using by checking if it is in the `scala.collection.mutable` or `scala.collection.immutable` package. You should always double-check to see which version you have imported!

Mutable collections can be changed (i.e. updated, added to, removed from) in place, meaning that you modify the original copy. Immutable collections cannot be modified in place, meaning that if you wish to change an immutable collection, you need to create a new instance.

For example, each time you add an element to an immutable hashmap, the add method returns a new instance of a hashmap, containing all the elements of the old hashmap plus the new element. As you can guess, immutable data structures can become inconvenient if you often modify their data. In your upcoming Search project, you will want to be sure that you use mutable hashmaps so that you can modify your data in place.

Should you declare your collection as a val or a var? The answer is found in the following table:

<table>
<thead>
<tr>
<th></th>
<th>Immutable Collection</th>
<th>Mutable Collection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>Yes! This is what you should use when you need to modify a collection. Because it is immutable, changing the contents won't change the original, so it's safer.</td>
<td>Don't do this — you can already modify the contents of a mutable collection, so you never need to reassign the var. You should instead make this a val or immutable.</td>
</tr>
<tr>
<td>Val</td>
<td>This is the safest! But won't allow you to change your data.</td>
<td>Sure! Always declare mutable collections as val. Be aware — other functions could modify the data! However, this is more space-efficient than an immutable structure.</td>
</tr>
</tbody>
</table>

Again: var is reassignable and val is **not** reassignable.

If you are wondering why a mutable collection can be declared as a val then read on! It turns out that when a val refers to a collection, it refers to a location in memory representing that collection. Because it is a val, not a var, the val can never be changed to refer to a different location in memory. However, the contents of that memory — the actual elements in the collection — are mutable, so the collection can still change!

For example, if you declare a mutable hashmap, say `val map1`, then `map1` refers to the hashmap, not the data contained within the hashmap. Because `map1` is a val, you cannot assign `map1` to refer to any other hashmap. However, because the hashmap itself is a mutable data type, you can change the contents of `map1` regardless. Because you can easily update val mutable collections, you should not ever use var mutable collections.

### 3 Team Rocket’s Shenanigans

Jessie and James have hijacked the system of a nearby Pokecenter and they want to change the address of all pokeball deliveries to Team Rocket’s headquarters! They decide to make a class containing a list of all the pokemons to transfer, but they aren’t quite sure whether they should use a val or a var in the class, and also whether the data structure containing the names should be mutable or immutable. Help them work out the pros and cons of the different implementations before they have to resort to mass transporting all the pokeballs with their not-so-trustworthy Meowth balloon (or other slow transit systems)!

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

**Take 1: Val Array**

Jessie decides to first try putting names in an array:
Array("Pidgey", "Piplup")

Recall that arrays are mutable. Jessie wants to make a Names class whose constructor takes in a private val array of pokemons to kidnap. Here is the class declaration:

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

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

**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 pokemon list, James figures that he might want to add more pokemons to his list in the future. He realizes 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 an address method that works with the list of names.

**Note:** Since Jessie and James do not want anyone else to see their list of pokemons and foil their plan, the List of pokemons is declared as a private var in the constructor:

```scala
class Names2(private var nameList: List[String])
```

**Note:** We had to use the private keyword in this class declaration, because if a class parameter in Scala is a var, the default for it is to be public.

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

**Take 3: Val Immutable List**

Jessie and James realize that they don’t want just anybody modifying the list of names. 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. Jessie and James need you to write address to return a new instance of Names3 with the correct internal list. Now their data is completely secure!

**Task:** Why do we now need to return a new instance of Names3, instead of having address 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.

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

### 4 Lunch Time!

It is lunch time and Brock is cooking up a storm with his new collection of yummy sandwiches! However, with Ash’s growing pokemon family, there are just too many mouths to feed. In this problem, you will use stackable modifications to help Brock make sandwiches. That is, you will write multiple traits, and then you will mix them in to make a variety of sandwiches.
**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 (generally) 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:
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", "Peanicles")

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

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

### 5 Just for Fun: Currying Foods (pun intended)

Ash, Misty and Brock are frustrated, because their regular favorite meals just don’t taste the same anymore! They refuse to eat any food item that has an inadequate tastiness ranking. They realize they need an automated way of sorting through foods based on their tastiness ranking, so they can still eat a good hearty meal. A Scala program should do it!

We represent a food using the following class:

```scala
class Foods(val name: String, val tastiness: Int)
```

We have written a function called goodFood, which takes a list of Foods and outputs a list of all the Foods that have more than 20 tastiness.

**Task:** Look at our implementation of this function. What would we need to do to our code if our heroes became very hungry and had lower standards for the tastiness of the food? Discuss this with your partner and write it down on a piece of paper.
The three realize that as they grow older, their food needs may change over time, but they don’t want to have to rewrite the code each time that happens. In comes currying! In Scala, a function can be defined with multiple parameter lists:

```scala
def addN(n: Int)(x: Int) = x + n
val add2 = addN(2)
println(add2(3)) // prints 5
```

If you call the curried function with only the first parameter list, it will return another function that takes in the second parameter list.

**Task:** Create a new file with a class in it called `Currying`. It should not take in any arguments in the constructor.

**Task:** Write a curried function `minTastiness`, which takes in a number `n` and a list of `Food`, and returns a list of all its elements with at least `n` tastiness.

**Task:** Now, curry `minTastiness` into a new function, `tastinessOver20`, which takes a list of `Food` and returns a list of all its elements with at least 20 tastiness.

**Task:** Write a curried function `sumTastiness`, which takes a number `n` and a list of `Food`. If the total tastiness in the list is at least `n`, it should return the original list. Otherwise, return a new (empty) `Food` list.

**Task:** Curry `sumTastiness` into a new function, `sumTastinessOver50`, which takes a list of `Food` and returns the list if the sum total tastiness is at least 50, and an empty list otherwise.

**Hint:** Your curried functions (`minTastiness` and `sumTastiness`) should look very similar to the function you looked at earlier that used hard-coded values.

**Hint:** To put our above example in the language of this exercise, `add` is a curried function that takes two numbers and outputs their sum. `add` is then curried into `add2`, which takes a number and adds 2 to it.

**Note:** As always, be sure to test all the functions you have written!

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](https://cs.brown.edu/courses/cs018/feedback)