Greetings - Scala fundamentals (part 4)

Aims

We're going to continue building our application that we started in *Greetings - Scala fundamentals (part 1)*.

First, we will refactor some of our code. Then, we will cover the following topics:

- Arrays
- Collections (List, Set, Map...)

Throughout this session, you can refer to the following helpful cheatsheet which is a nice guide to support you with the Scala syntax:
Scalacheat | Scala Documentation

Prerequisites

You have completed Greetings - Scala fundamentals (part 3).

Lesson steps

Currently we have a lot of classes / code inside our `GreeterApplication` and it's starting to become difficult to manage, it does not have a clear **separation of concerns**.

> **Separation of concerns** means, broadly, separating our code into distinct, separate parts. In our `GreeterApplication`, we have created a `Person` class. The `Person` class doesn't depend on the `GreeterApplication`, and it's possible that we might want to make use of `Person` again, in another part of our app.

We're going to refactor out classes into *packages*.

> A **package** is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software [...] can be composed of hundreds or thousands of individual classes, it makes sense to keep things organized by placing

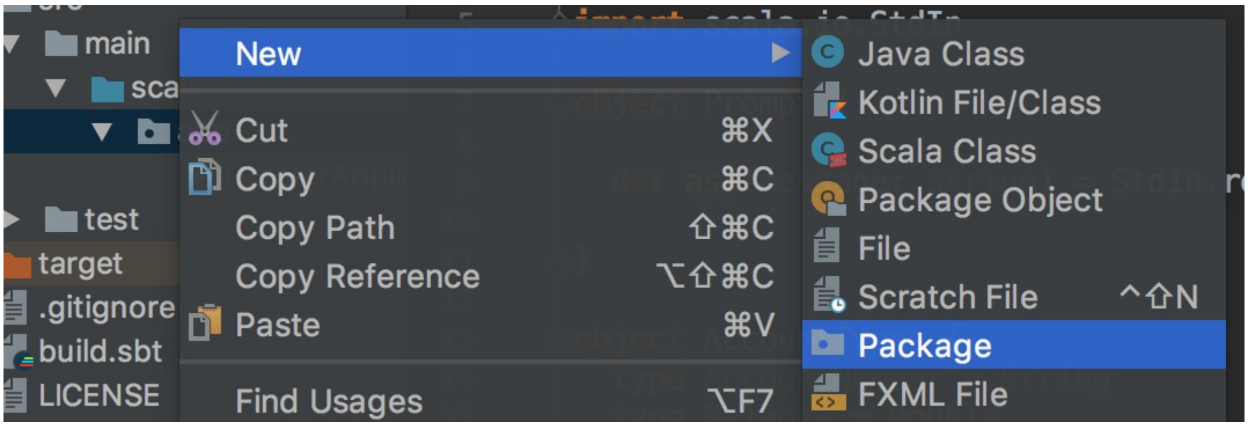
> related classes and interfaces into packages.
>
> From: What Is a Package? (The Java™ Tutorials > Learning the Java Language > Object-Oriented Programming Concepts)

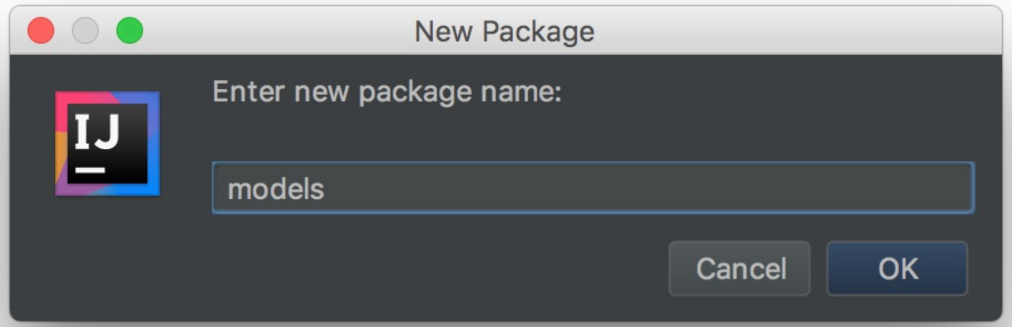
Let's create our first package called `models`. We're going to use the name `model` because our classes are *models* of data. We're also using this name because we are using a **design pattern** known as **Model-View-Controller (MVC)**.

> A **pattern** is like a blueprint that we can follow to put our application together. There are many architectural and design patterns, each of which is best suited to solving different problems.

We'll talk in more detail about patterns, and MVC specifically, later.

Within your `app` folder right-click and create a new package called `models`.



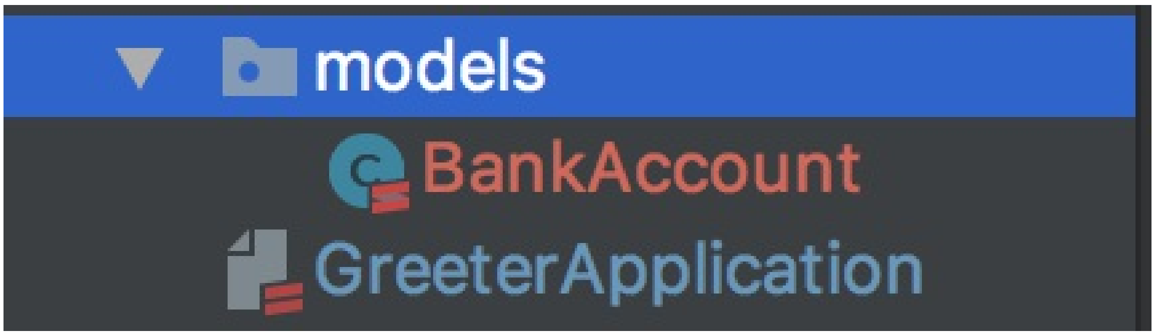


This will have a created a new directory within your `app` folder named `models`.

We can now *refactor* our *models* into this package. We will be moving our `Person`, `BankAccount` and it's subclasses into the models package.

We are now going to use a feature of IntelliJ IDEA to help us move our classes into the `models` package.
Right-click on your `abstract class BankAccount` and select *Refactor* -> *Move...* -> *To package* -> *app.models* -> *Refactor*.

You will now have moved the abstract class into the models package.



Your *BankAccount.scala* file should now reflect the following:

```
package app.models

abstract class BankAccount(accountNumber : String,
                           val balance: Double) {

  def withdraw(amount : Double) : BankAccount
  def deposit(amount : Double) : BankAccount

  override def toString: String = s"Account number:
      $accountNumber, balance : $balance"
}
```

Once this is done, if we open our `GreeterApplication` now, we might see something unexpected: IntelliJ providing us with warnings. Since we've moved our classes out into this own package, our `GreeterApplication` class no longer knows where to find them. We can find this using something called **imports**.

We can then correct the import statements at the top of our `GreeterApplication` to be the following:

```
import app.models.BankAccount
```

We can type this in manually, or select any of the code that is highlighted in red and push Alt + Enter.

As the word *import* suggests, our imports allow us to bring the `BankAccount` into our `GreeterApplication` class. The two classes remain separate and distinct, but our import statements act as a list of the related classes that `GreeterApplication` will require access to.

> An import is like a signpost for the compiler to use, that says: "When you encounter a need to use the `BankAccount` model, you can find it in `app.models.BankAccount` package."

This should resolve the issues caused by moving our abstract `BankAccount` into the models package.

Task ⏱

> - Refactor `final SavingsAccount` and `final CashISASavingsAccount` into their own files within the `models` package as you have done above
> - Refactor `class Person` into it's own file within the `models` package
> - Fix the import issues that are caused by moving our classes
>
> Once you have done this you can do the following to clean up your imports from the menu bar: *Code* -> *Optimize imports*. This will clean up the imports left over after you refactored the packages.

Now re-run your application to ensure the code still compiles and runs successfully.

Hooray! 🎉

Refactoring Prompt

We're going to refactor our `Prompt` object slightly to implement a `reply()` method.

Ensure your Prompt object looks like the following:

```
object Prompt {
  def ask(message: String) = StdIn.readLine(message)

  def reply(message: String) = println(message)
}
```

This would be known as your `View` in the MVC paradigm, it is responsible for the representation of your model. Again, more on this later!

Task ⏱

> Create a new package inside your *app* directory called *views* and refactor your Prompt class to be inside this newly created package.
>
> Finally, refactor your `GreeterApplication` to make the following call instead `Prompt.reply(loyal.speak())`.
>
> Ensure your app still compiles and runs!

Congratulations, you refactored your app and started to implement the MVC pattern.

☕ Grab a cup of tea!

Collections

In almost all programming languages you will come across various collection data structures. Collections represent, as the name suggestions, a collection of data.

These collections can take many different forms and shapes. The most comment collection types that you will come across in most languages are:

- Arrays
- Lists
- Sets
- Maps

In particular for Scala for will also come across:

- Tuples
- Options

We are going to cover the basics of Arrays, Lists, Sets and Maps.

An exhaustive list of Scala collections can be found here:
Mutable and Immutable Collections | Scala Documentation

Arrays

Scala provides a data structure called an `Array`, an Array stores a fixed number of elements in a sequential order where each element is the same type. i.e. `Int`, `Boolean`, `String` or `Person` for example. You use an Array to store a collection of data.

The syntax for creating an `Array` in Scala is:

```scala
val names = Array("Adam", "Daniel", "John")
```

This will have declared the *immutable* value *names* as type `val names : Array[String]`.

If you know the length that your Array must be before you know the values, you can also declare an Array in the following way:

```scala
val names = new Array[String](3)
```

This is stating that we want to create an Array that will contain 3 Strings. This is useful if you're trying to do memory optimisations! But for now, I wouldn't worry about that *too much*.

Lists

Another data structure that Scala provides is `Lists`. Similar to `Arrays`, `Lists` in Scala store a collection of data in a sequential order, however, there is an important difference!

Remember the tip above when `Arrays` are quick if you are accessing an element at a random index? Well, In Scala, a `List` has better performance if you are accessing the `head` element in the List, rather than randomly accessing the element at an index.

🤔 That's enough technicalities for today! Just keep in mind that some data structure perform better than others in certain scenarios.

The syntax for creating a list in Scala is:

```scala
val names = List("adam", "david", "john")
```

We can also create a `List` in Scala by concatenating multiple elements together:

```scala
val names = "adam" :: "david" :: Nil
```

This will give us a `List` as in the example above. Did you notice `Nil`?

`Nil` in Scala is a *subclass* of `List`, this means it *extends* the List class and provides a default implementation which is empty. In short, `Nil` is an empty List!

We could have done the following:

```
val names = Nil
```

This would mean that our names *immutable* value does not have any elements in the list, yet.

Both `List` and `Array` extend from the `Iterable` interface - meaning that we can **iterate** over them. If we **iterate** over a collection, we step through each element in the collection one-by-one.

Methods available on `Iterable` data types

There are **many** methods that you can call on a collection that inherits from `Seq`, these include:

- `head` gets the first element in the Array
- `tail` gets all the elements except the first in the Array
- `length` / `size` returns how many elements are in the Array
- `reverse` creates a new Array with the elements in reverse order

Exclusive to Arrays:

- `update()` updates an element at a specific index with a new value (the immutable Array creates a new Array with the new value at that position)

There are many more!
Scala Standard Library API (Scaladoc) 2.10.4

Accessing an item at a specific position in a collection

You can access an element at a specific index/position in any collection that *inherits* from the `Seq` (Sequence) interface/class.

You could do the following to return the first element in the Array/List:

```
val names = Array("Adam", "Daniel", "John")
val adam = names.head

OR

val names = List("adam", "daniel", "john")
val adam = names.head
```

We could also return an item at a specific index:

```
val names = Array("Adam", "Daniel", "John")
val daniel = names(1)

OR

val names = List("adam", "daniel", "john")
val daniel = names(1)
```

You can also call these methods on other implementations of `Seq` such as `Lists` , `Range` , `Vector` ...

When operating on `Lists` and `Arrays` or any *sequential* data structure in Scala it is important to remember that the index of the first element *(head)* in the list is **always** zero. The index of the last item in the list is **always** the total number of elements in the list minus one.

> An **index** is a number that corresponds to the location of an **element**, or a space where an element might be. An **element** is the thing that is stored inside of the collection.

Tip 💡

> Arrays are good to use when you are going to be accessing elements at random positions! For example; you have an Array of 1000 elements and you retrieve the element at position 500. This is very fast with an Array, not so much if you used a List.

Further reading
Mutable and Immutable Collections | Scala Documentation

Implementing a List

Enough technical gibberish! Let's get a real-world example implemented into our Greeting Application.

We're going to refactor our implementation of how we do our conditional expression `if (name == "adam") { ... }`.

Instead of checking for a single name we are going to implement a `List` of names. We will check if the name is an element of this list.

Inside of our `Person` class we are going to want to implement the following:

```
private val excluded = List("adam", "daniel")
```

We can then refactor our implementation of our `speak()` method to check if our `Person` name is an element in the *excluded* list.

```
def speak() : String = {
    if (excluded.contains(name)) {
      s"You don't get a hello!"
    } else {
      s"Hello $name, you are $age
      $years old. \nYour account details are: $bankAccount"
    }
  }
```

Re-run your application and check it still compiles and runs. Try checking both names in the list!

🕑 Don't forget to commit your code.

Sets

Scala has another collection called `Set`.
`Sets` are similar to `Lists` in that they are a collection of elements, however, there are some important differences.
Sets are a collection of pairwise different elements of the same type. i.e. `Int`, `Boolean`, `String` or `Person` for example.

Tip 💡

> Sets cannot contain duplicate elements, this is a great way if you need to ensure the elements are unique.

We can declare a Set in the following way:

```
val names = Set("adam", "daniel", "david")
```

We cannot access a specific item at a position like we can with `Array` and `List`. This is because a `Set` is not an ordered collection. However, we can use the same methods as previously described: `map`, `filter` `head`, `tail` etc.

We can also iterate over a `Set` to look at each individual element.

Task ⏱

> Try to implement a `Set` which has a duplicate element, run your application and see what happens.

```
val nameSet = Set("adam", "daniel", "david", "adam")
println(nameSet)
```

Did you notice how it removed the duplicate *adam* element from the `Set`. This didn't throw an exception at runtime either!

There are also useful methods available to `Set`. If you have a `Set[Int]` then you can call `min()` or `max()` to determine what the minimum and maximum number is in the collection.

Maps

Scala has another collection called `Map`.

`Maps` are not similar to `Lists` or `Arrays` in Scala as they are a key, value pair of elements. This means that each element has a key, which has an associated value assigned. Each key in a `Map` **must** be unique. However, you can have duplicate values across the keys.

In a sense, you are assigning your own *index/keys* to `Map`, rather than using a sequential order which is zero based.

The syntax for creating an empty Map is:

```
val fruits : Map[String, String] = Map()
```

We can create a Map of fruits, where the key is the name of the fruit and the value is the colour of that fruit.

```
val fruits : Map[String, String] = Map(
"orange" -> "orange",
"banana" -> "yellow",
"apple" -> "red"
)
```

Tip 💡

> Remember that all the keys need to be unique, therefore, we could not have another element of "apple" -> "green".

In order to access an element in a `Map`, we have to call our Map with parenthesis `(argument)` where the argument is the *key*.

```
val fruits : Map[String, String] = Map(
"orange" -> "orange",
"banana" -> "yellow",
"apple" -> "red"
)

val colour : String = fruits("banana")
```

This would set the *immutable* value `colour` to the value *yellow*.

- We can list all of the *keys* in a `Map` by calling `.keys()`
- We can list all of the *values* in a `Map` by calling `.values()`

We can also check if our `Map` is empty by calling `.isEmpty()`.

Implementing multiple Bank Accounts

In this section, we're now going to make each *instance* of the `Person` class have multiple `BankAccount` implementations.

i.e. a Person can have a `SavingsAccount` and a `CashISASavingsAccount`.

Change the class declaration of `Person` to be the following:

```
class Person(name : String, age : Int,
  val bankAccounts: Seq[BankAccount] = Nil) { ...
```

As you can see, we have now said that each Person will have zero or more bank accounts. We have indicated that if we do not set a value for the argument *bankAccount*, then the default argument will be an empty List `Nil`.

Note 📋

> Did you notice how we used a Seq rather than a List when declaring our Person interface? Referring to the type inheritance diagram above, why do you think this is?

Task ⏱

> Since we have changed our constructor for our `Person` class. How can we resolve the issue and instantiate an instance of Person where it is now expecting a `Seq` of BankAccount.

Discuss your decision on how you implemented this with your colleagues.
Re-run your application to ensure it still compiles.

The simplest fix for our compilation issue would be to instantiate our `Person` class in the following way:

```
val person = new Person(name, age.toInt, Nil)
```

This will create a `List` of `BankAccount` with a single element/account.

Remember that we are going to have to fix our secondary constructor as well.

Re-run your app and see what the output is now.
⏱ Don't forget to commit your code.

You will have seen that our `speak()` method now outputs `"Your account details are: List()"` this is because we do not have any accounts.

List containing a single Bank Account

The simplest way of instantiating a Person with a single bank account is to change our implementation to the following:

```
val person = new Person(name, age.toInt, List(withdrawn))
```

You should see the following when we run our app `"Your account details are: List(Account number: 45676, balance : 1000.0)"`.

However, we could have been more type specific and implemented either the following:

```
val person = new Person(name, age.toInt, IndexedSeq(withdrawn))
```

Or:

```
val person = new Person(name, age.toInt, LinearSeq(withdrawn))
```

Discussion

> What are the benefits of either solution? Work in pairs to discuss the benefits of either referring to the link below and present your findings back.

Scala Standard Library 2.12.3 - scala.collection.IndexedSeq

Scala Standard Library 2.12.4 - scala.collection.LinearSeq

In short, implementing a `IndexedSeq` is better if we're accessing `BankAccount` at a random index. i.e. we are withdrawing money from an account at an ATM.
Implementing a `LinearSeq` would be better if we're operating on all of the accounts for that person. i.e. totalling the funds in each account.

Recap

We covered the following topics:

- Collections (Array, List, Set, Map) and Iterable interface
- We have talked around the benefits of various collections, their performance implications
- Operators
- Underscore, an example with iterating

Resources

- Scalacheat | Scala Documentation
- What Is a Package? (The Java™ Tutorials > Learning the Java Language > Object-Oriented Programming Concepts)
- Mutable and Immutable Collections | Scala Documentation
- Scala Standard Library API (Scaladoc) 2.10.4
- Mutable and Immutable Collections | Scala Documentation