Greetings - Scala fundamentals (part 6)

Aims

We are going to continue to cover the fundamentals of Scala and continue to build the application we built in *Greetings - Scala fundamentals (part 2)*.

We will be covering the following topics:

- Tuples
- Traits
- Case classes
- Pattern matching
- Companion Objects
- Options
- Functional methods (flatMap…)

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 6).

Lesson steps

Tuples

Scala has another collection named a *Tuple*. A tuple combines a fixed number of items together so that they an be passed around as a single object.

> Unlike an array or list, a tuple can hold objects of different types, however, they are still immutable

An example of a tuple is:

```
val t : (Int, String, Person) = (1, "hello", new Person())
```

As you can see, this has created a tuple containing 3 very different data types, where as with a List, we would have to have a common interface i.e Iterable, Person or another common type.

Tip

> There is currently an upper limit of the number of objects a tuple can hold, this is 22. If you need anything beyond that then you need to use a collection.

You can then access the elements in the tuple using the following syntax:

```
val t : (Int, String, Person) = (1, "hello", new Person())

val number = t._1
val string = t._2
val person = t._3
```

Just like Iterable collections, you can iterate over a tuple:

```
val t : (Int, String, Person) = (1, "hello", new Person())

t.productIterator.foreach(x => println(x))
```

The above example would iterate over each element and print it to the console.

Traits

We may have instances when we want to share interfaces and fields between classes. Earlier, we looked at how we could use Abstract Classes and inheritance to share behaviour across multiple subclasses. Now, we're going to look at another way of defining a shared set of behaviour, using something called a Trait.

We can create a basic trait in the following way:

```
trait Pet
```

As with Abstract Classes, it's not possible to instantiate a Trait. Instead, we use the Trait to establish method and field definitions, which can then be extended by other classes. This is also referred to as 'mixing' the trait in.

Tip

> Traits can not have constructor parameters like abstract classes can.

```
trait Pet {
    val name: String
}
```

Above, we have introduced a field definition to our Pet Trait. This means that any Class which extends our Trait must implement a name field.

```
final class Cat(override val name: String) extends Pet
final class Dog extends Pet
```

We have defined two additional Classes that extend our new Trait.

Tip

> You can 'mix in' multiple traits, by using the `with` keyword, whereas you can only ever extend one abstract class::

Do you notice anything unusual about the Class Dog? IntelliJ has highlighted to us that Dog Class either must implement the name field, or it cannot extend the Pet trait. Fix the Dog Class, so that it also implements the name field.

```
final class Cat(override val name: String) extends Pet
final class Dog(override val name: String) extends Pet
```

We can now consider these to be Subtypes of our trait Pet. This means that any time the compiler expects us to provide a Pet type Object, we can provide a Cat or Dog, or both.

```
private val _pets : ArrayBuffer[Pet] = new ArrayBuffer[Pet]()
```

Though it's not possible to instantiate a Pet, we can use declarations like the above to create a collection - in this case an `ArrayBuffer` - that can contain any Object that is a Subtype of Pet. Our pets collection could therefore contain a mixture of cats and dogs, and we could be confident that if we tried to access the names of these pets, then each one - regardless of whether it was a Cat, Dog or other subtype of Pet - would have a name we could access.

Further reading
Scala Tutorial - Learn How To Use Scala's Mutable ArrayBuffer

If we wanted to, we could provide an implementation in our Pet trait, which would then be automatically shared by all of the extending Classes.

Tip

> Traits can have a partial implementation.

```
trait Pet {
    val name: String
    def speak: String = "Woof!"
}
```

Now, we've provided a function that any Class extending the Pet trait can access, though we might decide to override this in our child Classes.

When should we use a Trait?

> The main case when we would want to use a Trait is if the behaviour it defines would be used in multiple, unrelated classes.
>
> As with Abstract Classes, we would not want to use a Trait if the behaviour it defines will not be reused. Alternatively, if we wanted to establish inheritance of a fixed, shared set of behaviour, it would be better to use an Abstract Class.

Task

> 1. Implement a new file inside of your `models` package named *Pet.scala*
> 2. Inside *Pets.scala* implement a `trait Pet` which has:
>   i. An immutable property *name* which is of type `String`
>   ii. An method `speak()` which returns a `String`
> 3. Implement **two** classes (Dog and Cat) which extends the `Pet` trait / interface
> 4. Provide the implementation for the *name* property and `speak()` method
> 5. A dog, should *woof!*

> 6. A cat, should *meow…*

Example implementation
Pet example · GitHub

Referring to the example implementation, did you notice the `sealed` keyword?

> `sealed` is another Access Modifier that we discussed in Part 3, when referring to traits; it means that we can only extend a trait within the same Scala file.
> i.e. a Person cannot be a Pet…

Further reading
scala - What is a sealed trait? - Stack Overflow

To put this in context, we are going to adopt a design pattern known as a *Repository*. We are going to implement the following:

1. PetRepository that holds a `ArrayBuffer[Pet]`
2. Has the following methods:
    i. `def all() : List[Pet] = Nil`
    ii. `def findByName(name : String) : Option[Pet] = None`
    iii. `def dogs() : List[Pet] = Nil`
    iv. `def cats() : List[Pet] = Nil`
    v. `def other() : List[Pet] = Nil`
    vi. `def add(pet : Pet*) : List[Pet]`
        a. ::**Note**: The asterisks is syntax for varargs::
    vii. `def removeByName(name : String) : List[Pet]`
    viii. `def update(pet: Pet) : List[Pet]`

Tip

> To achieve the above, you're going to want to create a new package called repositories at the root of the project, containing a PetRepository.scala file.

Task *30 minutes, paired programming*

> Implement the above. Good luck!

Don't forget to commit your code.

Consider the following:

- Do you need to have multiple instances of `PetRepository` or would a singleton suffice?

- Does the `PetRepository` need to be mutable and maintain state?

Further reading
Scala varargs syntax (and examples) | alvinalexander.com

Example implementation
Pet Repository · GitHub

Case classes

Part of the functionality of our `PetRepository` involves being able to get a `List` of a particular type of pet, using our `cats()` and `dogs()` methods. One way that we can do this is through something called **pattern matching**. The `case class` is particularly effective at pattern matching, so let's refactor our `Pet` subclasses to make use of this functionality.

A `case class` is a type of class, which we can define as follows:

```
final case class Cat(val name: String) extends Pet { ... }
final case class Dog(val name: String) extends Pet { ... }
```

Whenever we want to create a `class` with immutable state, we should use a `case class`. Generally speaking, we should also use a `case class` when we don't want any logic within our `class`, though it *is* possible to define methods within a `case class`.

By defining a `case class`, we get access to default implementations of the `equals` and `toString` methods. The `equals` method allows us to compare instances of the `case class` based on their structure, rather than their reference. This is known as structural equality.

Case classes also give us two extremely useful methods, `apply()`, `unapply()` and `copy()`.

> The `apply()` method can be though of in the following way: Given a set of arguments, we want to *apply* those to our constructor and generate an Object of that type.

> The `unapply()` method takes an instance of a Class and deconstructs it to expose it's public properties as a Tuple.

```
val c1 = Cat("Felix")
val c2 = Dog("Spot")
val c3 = Cat("Felix")

c1 == c2 // false
```

```
c1 == c3 // true
println(c1.name) // "Felix"
```

It's possible that `c1` and `c3` are two unique cats, meaning they refer to different allocations of memory. They will occupy two different reference points, but now that `Cat` is a `case class`, the fact that they have the same name makes them equal. We probably need to introduce some way of distinguishing between the two of them, but we'll think about that later…

We also ensure that the attributes of our cats and dogs are immutable. We previously had to declare this explicitly, by using the keyword `val`, but the `case class` implementation will ensure this for us. Based on this, what do you think would happen if you tried:

```
c1.name = "Tom"
```

Did you think it wouldn't compile? That's right.

Did you notice anything else different about the `case class` definition of `Cat`, when compared to the `class` definition `Cat`? The `override` key word is no longer needed.

Further reading
Case Classes | Scala Documentation

Task

> Refactor your class to use the `case` keyword.

Now that we've refactored our `Pet` subclasses, let's see how we can benefit from the **pattern matching** functionality that they provide.

> Don't forget to commit your code.

Companion objects

An `object` with the same Name as a `class` is known as a *companion object*. A companion object can have access to the private members of it's companion. An example of this could be:

```
class Dog private(name : String, private val personYears : Int) {
  import Dog._
  def age = calculateAgeInDogYears(personYears)
}

object Dog {

  def withName(name : String) = new Dog(name, 2)

  private def calculateAgeInDogYears(age : Int) = age * 7

}

val d = Dog.withName("bruce").age
```

Here we have created a private constructor for the Dog. Therefore we cannot do `new Dog()` *outside* of the companion object. However, we can instantiate Dog within the companion object. You will have noticed as well that we can call the `calculateAgeInDogYears` function from within the `Dog` class. We have done this by importing the members from `Dog` object inside of the class. `import Dog._`.

We have also introduced what is known as the Factory Pattern. By specifying a `private` constructor on `class Dog`. We are only able to instantiate an instance of `Dog` via it's companion object, otherwise known as Factory. This is done by calling the factory method `def withName(...)`.

Factory methods are particularly useful for providing a concise interface for creating classes.

Scala objects, like case classes also come with `apply()` and `unapply()` methods. This allows us to instantiate an object, or to deconstruct an object.

Taking the above code example with `Dog.withName`. If we were to remove the private keyword from the `Dog` class:

```
class Dog(name : String, private val personYears : Int) {

  import Dog._
  def age = calculateAgeInDogYears(personYears)
}

object Dog {

  def withName(name : String) = new Dog(name, 2)

  private def calculateAgeInDogYears(age : Int) = age * 7

}
```

We can then instantiate a Dog using the `apply()` method of the companion object.

```scala
val d = Dog("bruce", 1)
```

This is the equivalent of writing `Dog.apply("bruce", 1)`.

Further reading
Companion objects

Pattern Matching

Pattern matching is a mechanism for checking a value against a pattern. A successful match can also deconstruct a value into it's constituent parts.

```scala
val number = 1

def intToString(n : Int) : String = {
    n match {
        case 0 => "zero"
        case 1 => "one"
        case 2 => "two"
        case _ => "above two"
    }
}

intToString(number) // returns "one"
```

In the example above, you can see that we have pattern matched over our value, and deconstructed this into it's parts. In this instance, a single primitive type `Int`.
We have then evaluated an expression and returned a `String`. The last case statement, introduces another use of the `_` operator. This time it's referring to *any other value*. If we did not use the `_` operator, this would have thrown a `scala.MatchError` exception. This is because we haven't accounted for every possible value that `n` could have been.

Pattern matching can also match on the *Type* of an object.

```scala
def isDog(pet : Pet) : Boolean = {
    pet match {
        case d : Dog => true
        case c : Cat => false
    }
}
```

```scala
val d = Dog("geoff")
isDog(d) // returns true
```

Task

> Why do you think we didn't use the `_` operator in the above example?

The reason we didn't use the `_` operator is because the Scala compiler knows that we have accounted for every subclass of `Pet`. We have told it this by using the `sealed` keyword on our `Pet` trait, and every possible concrete implementation is in the same file. Whereas, the example above, an `Int` could be any value.

We can also pattern match on a `case class`:

```scala
sealed trait Pet {
  val name : String
  def speak() : String = "hello"
}

final case class Dog(override val name : String) extends Pet {
  override def speak(): String = "woof!"
}

final case class Cat(override val name : String) extends Pet {
  override def speak(): String = "meow..."
}

def whoDis(pet : Pet) = {
    pet match {
        case Dog(name) => s"This is a dog, and their name is $name"
        case Cat(name) => s"This is a cat, and their name is $name"
    }
}

val d = Dog("geoff")
whoDis(d) // returns "This is a dog, and their name is geoff"
```

Let's add another property to our `Pet`. We're going to store their age.

```scala
sealed trait Pet {
  val name : String
  val age : Int
  def speak() : String = "hello"
}
```

```
final case class Dog(override val name : String, age : Int) extends Pet {
  override def speak(): String = "woof!"
}

final case class Cat(override val name : String, age: Int) extends Pet {
  override def speak(): String = "meow..."
}
```

Now that we have multiple properties, we can refactor our method above to *only* be concerned about their `name`.

```
sealed trait Pet {
  val name : String
  val age : Int
  def speak() : String = "hello"
}

final case class Dog(override val name : String, age : Int) extends Pet {
  override def speak(): String = "woof!"
}

final case class Cat(override val name : String, age: Int) extends Pet {
  override def speak(): String = "meow..."
}

def whoDis(pet : Pet) = {
    pet match {
        case Dog(name, _) => s"This is a dog, and their name is $name"
        case Cat(name, _) => s"This is a cat, and their name is $name"
    }
}

val d = Dog("geoff", 3)
whoDis(d) // returns "This is a dog, and their name is geoff"
```

Here we have another example of the `_` operator. In this instance we are saying when we deconstruct our concrete `Pet` we do not want to assign their age to a variable, in essence it is ignored. We're effectively saying that we don't care what the value assigned to `age` is, because we aren't going to use it as part of the return value of `whoDis()`.

With pattern matching, we can also apply a guard (if statement) for each case statement. This is useful if we want to determine if a case passes a simple boolean statement.

```
sealed trait Pet {
  val name : String
  val age : Int
  def speak() : String = "hello"
}

final case class Dog(override val name : String, age : Int) extends Pet {
  override def speak(): String = "woof!"
}

final case class Cat(override val name : String, age: Int) extends Pet {
  override def speak(): String = "meow..."
}

def whoDis(pet : Pet) = {
    pet match {
        case Dog(name, age) if name.equalsIgnoreCase("geoff") =>
            s"""This is top dog,
          my name is $name, give me your lunch!
            I am the eldest at $age years old.""".stripMargin
        case Dog(name, _) => s"This is $name, I'm only a dog."
        case Cat(name, _) => s"This is a cat, and their name is $name"
    }
}

val geoff = Dog("geoff", 10)
val barry = Dog("barry", 2)

whoDis(barry) // returns "This is barry, I'm only a dog."
whoDis(geoff) /* returns "This is top dog, my name is geoff, give me your lunch!
                                I'm the eldest at 10 years old." */
```

The example above has introduced a guard for the first case statement, this checks if the value at `name` equals *geoff*. If so, this case matches and it returns the top dog string.

Further reading
Pattern Matching | Scala Documentation

> Don't forget to commit your code.

Options

One of our `PetRepository` functions has a return type of `Option[Pet]`. Most of our default method implementations returned `Nil` - an empty `List` - but this method returns something different: `None`. As the name suggests, an `Option` represents an optional value. If there is no value, this is represented as `None`, in the same way that `Nil` represents a `List` containing no values. If there is a value, this is represented by `Some(x)` where `x` is the value.

> The `Option` wraps the value, or lack of value, inside of a container. There may or may not be something inside of the container. We won't know until we open the container up. When we access what's inside an `Option`, we'll only ever find one or zero values. In this way, it's a bit like a `List` or `Map`; it might be empty, or it might have something

> inside of it.

Can you think why we might want to use a `Option[Pet]` as the return type of our `findByName(name : String)` method? In this method, we always want to return a `Pet` - but what if there is no `Pet` with a name matching our argument? There is no generic `Pet` that can be returned…

This is where the `Option` becomes useful. We can say that if we find a `Pet` matching the `name` argument, we return it as `Some(pet)`. If we can't find a `Pet` matching our `name` argument, then we return a `None`.

In most programming languages, this is normally represented as a `null` value.

Tip

> Scala is a type safe language, therefore we can make guarantees on the types returned and passed to functions. null would break this guarantee. In Java, it is common to handle NullPointerExceptions, this is because it isn't type safe; therefore there is no guarantee a object is instantiated.::
>
> `None` tells the compiler, program, that there is no value, however, None is a value with a memory pointer that can be passed around to represent that. `null` is not.

Further reading
[What is null in Java]

```
val c: Option[Pet] = Some(Cat("Tom", 3))
val n: Option[Pet] = None
```

Above, we can see how this might look in our code. We can treat `c` and `n` as both being of the same type, `Option[Pet]`. This means that any method requiring an argument of `Option[Pet]` would accept both `c` and `n` - although there is technically no `Pet` inside of `n`.

> The `Option` has a number of useful methods, including `isEmpty()`, which would tell us that our `n` value above is empty and does not contain a value. The opposite of this is `isDefined()` which tells us if the `Option` wrapper contains a value, i.e. Some(x). If we know that there is something inside of the `Option`, we can call `get()` to turn it from an `Option[Pet]` into a `Pet`.

What might happen if we call `get()` on our `c` and `n` values above? Because we've defined these as `Option[Pet]` type, the compiler expects that a `Pet` will be the return type of `get()`. But what happens if the `Option` actually contains a `None`?

Tip

> You can also map over an Option[T]::
>
> ```
> val x: Pet = c.get // returns Cat("Tom", 3)
> val y: Pet = n.get // returns NoSuchElementException
> ```

Both of the above could be passed into any method that requires a `Pet`, but trying to call `y.name` would obviously be impossible! This shows one of the potential pitfalls of using an `Option` and is something to be mindful of when we're writing our code.

Further reading

Scala Options

Let's implement a concrete example of how to use `Option[T]` inside our `PetRepository` class.

Here's an implementation for our `findByName(name : String)` method:

```
def findByName(name : String) : Option[Pet] = _pets.find(p: Pet => p.name == name)
```

Let's look at what's happening in this method.

1. We use the `_pets` collection and call a method called `find()`
2. `find()` returns the first element in a collection that matches a predicate function
3. Our predicate function in this case is `p: Pet => p.name == name`.
   - `p` represents our iteration over the `_pets` collection.
   - We use the `=>` operator to indicate that we are going to evaluate `p` in the following function.
   - We then evaluate `p.name == name`, comparing each element in our `_pets` collection against the `name` argument in the `findByName` function.
4. Once our predicate is satisfied, we return an `Option[Pet]`.
   - If we couldn't find a match in our collection, this will contain a `None`.
   - If we found one (or more) matches, we will return the first matching `Pet`, in the form `Some(pet)`.

Don't forget to commit your code.

Functional methods (flatMap...)

In our implementation of `cats()` and `dogs()` we currently return the entire `List` of `Pet` instances, without distinguishing between the subtype of `Pet`.

```
def cats: List[Pet] = _pets
```

Instead, we want `cats()` to only return a `List` containing instances that are of type `Cat`. We can start by changing the return type of the function.

```
def cats: List[Cat] = _pets
```

At the moment, we don't have any pets in our `_pets` collection - but what do you think would happen now if we had a `List` that contained only a single `Dog`? We should see a warning in IntelliJ, letting us know that the return type of our function did not match the actual type of `_pets`.

Implement `cats()` method using pattern matching as shown below.

```
def cats : List[Cat] = {
    _pets.toList.flatMap {
        case c @ Cat(_, _) => Some(c)
        case _ => None
    }
}
```

Let's break this down.

1. We have declared our method `def cats : List[Cat]` to return a `List` of `Cat`
2. We then taken our mutable `_pets` `ArrayBuffer`, iterate over it with `.toList` to create an immutable `List`
3. We iterate over this using the `flatMap` function.
4. `flatMap` is similar to `map` that we have discovered earlier, with a clear difference
5. We then pattern match each item in the list, ensuring it is a `Cat(_,_)` and store the cat in a variable with `c @`. This will make our whole Cat object available at `c`
6. If the current element is a `Cat`, we then return an `Option[Pet]` to the list, which is create with `flatMap`

7. If the current element is not a `Cat`, we then return `None` to the list
8. We will have generated the following `List(Some(Cat), None, None, Some(Cat))`
9. `flatMap` then traverses the list and filters our all occurances of `None`
10. We end up with `List(Cat, Cat)`
11. `flatMap` has flattened the list and un-wrapped the Options

We haven't yet implemented all the methods in our `PetRepository`.

Task *2 hours, group exercise*

> Research and implement possible implementations to the following:
>
> 1. `def update(pet : Pet) : List[Pet]` - Modify the list for a single `Pet` and return the list in it's modified state
> 2. `def add(pet: Pet*) : List[Pet]` - Add a variable amount of `Pet`s to the `List`, return the modified list in its modified state, don't forget to update the list in the repository
> 3. Implement the `dogs()` method to return a `List[Dog`
> 4. Implement the `other()` method to return a `List[Pet]` that aren't `Dog` or `Cat`
> 5. Add multiple instances of a `Pet` to our `PetRepository` using the `add()` method. Once added, test the `findByName(name : String)` method to return both a `Some(_)` and `None`
> 6. How could we solve the problem where multiple `Pet`s have the same name? We're using the `name` as a unique identifier at the moment.

After you have researched and implemented the solution in your groups, present back to the academy with your proposed solution, and be able to reason about your decision.

Use the Documentation | Scala Documentation, Scala Standard Library 2.12.4 or StackOverflow to support your research.

> Don't forget to commit your code.

Recap

To recap, we have introduced a number of concepts in this part:

- Tuples
- Case classes
- Traits
- Pattern matching
- Options

- Functional method (flatMap)
- `_` operator
- NullPointerException

Resources

- Scalacheat | Scala Documentation
- Scala Tutorial - Learn How To Use Scala's Mutable ArrayBuffer
- scala - What is a sealed trait? - Stack Overflow
- Scala varargs syntax (and examples) | alvinalexander.com
- Case Classes | Scala Documentation
- Pattern Matching | Scala Documentation
- What is null in Java
- Scala Options
- Documentation | Scala Documentation
- Scala Standard Library 2.12.4