Greetings - Scala fundamentals (part 3)

Aims

This session will cover the following topics:

- Named arguments / default arguments
- Access modifiers
- Secondary constructors
- String interpolation

We will be expanding on the application that we built in *Scala fundamentals (part 2)*. We will be adding a SavingsAccount to each person as a default.

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 2).

Lesson steps

Let's set up every Person to have a SavingsAccount by default

We're now going to modify our `Person` class to have a `BankAccount` class which is private to that person. This means that only that person will be able to access the methods on their `BankAccount`, `balance`, `withdraw` etc.

In order to achieve this, we're going to have to add a `BankAccount` as a property to the `Person` class but make sure this is private!
The signature for our Person class will now become:

```
class Person(name : String, age : Int, private val bankAccount: BankAccount) { ... }
```

IntelliJ IDEA will now give you a warning where you try to instantiate a new Person. This is because it we are not providing a BankAccount to that Person in it's constructor.

Rather than changing how we instantiate a `Person` and pass a SavingsAccount as an argument, we're going to make it so that every person is provided with a default `SavingsAccount`. This can be done by providing a **secondary** constructor to the `Person` class.

Further reading

How to create multiple class constructors in Scala | alvinalexander.com

The first line inside of our `Person` class implementation should become:

```
def this(name: String, age : Int) = this(name, age,
                        new SavingsAccount("12345", 0.00))
```

Full example:

```
class Person(name : String, age : Int, private val bankAccount: BankAccount) {

  // secondary constructor
  def this(name: String, age : Int) = this(name, age,
                        new SavingsAccount("12345", 0.00))

  private def years : String = if(age > 1) "years" else "year"

  def speak() : String = {
    if (name == "adam") {
      s"You don't get a hello!"
    } else {
      s"Hello $name, you are $age years old"
    }
  }
```

What we have done here is provide a default value for the `bankAccount : BankAcccount` property in the case we create a person *without* this argument. Such as `new Person(name, age.toInt)`. This means that every Person is going to be provided with a default Savings Account.

> **Slightly** worrying that every persons bank account is going to have the same account number at the moment!

Now, in order to determine how much is in their account we're going to have to implement this within our `speak()` method.

Partially change your speak method implementation to be:

```
s"Hello $name, you are $age $years old. \n
  You have ${bankAccount.balance} in your account."
```

We have used the `${}` syntax to tell Scala we're going to evaluate expressions within this String, in this instance, value of a property.

At the moment, the `balance` method will be red in IntelliJ IDEA, to fix this we're going to make the balance a *public* property.

All properties on Scala Classes are *protected* by default, except for `case class` which we will visit later.

Change our BankAccount class to have the following signature:

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

This will resolve your issue and allow the `Person` `speak()` method to call their `BankAccount` balance.

Commit your code!

Now let's expand our `BankAccount` so we can `println` more information than just the balance.

Implement the following method inside of your `BankAccount` class:

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

And now make your `speak()` method the following:

```
s"Hello $name, you are $age $years old. \n
```

```
    Your account details are: $bankAccount"
```

This will print out the following:

```
What is your name? daniel
How old are you? 10
Hello daniel, you are 10 years old.
Your account details are: Account number: 12345, balance : 0.0
```

Here we have overridden the `toString()` method which is called where we want to turn out `BankAccount` into a string; this is useful for logging purposes and others.

Implementing CashISASavingsAccount

Let's modify our `GreeterApplication` so that we instantiate an instance of `CashISASavingsAccount()` and we assign this as the `Person` *bankAccount* argument. This means that our Person will have a CashISASavingsAccount rather than a standard savings account, this means we can deposit, but not withdraw from the account.

```
    val cashisa = new CashISASavingsAccount("45676", 0.0)
    val deposited = cashisa.deposit(1000.00)
    val withdrawn = deposited.withdraw(200.00)

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

Re-run your application, you will notice that the Persons balance remains at £1000.00. This is because we have provided an alternative subclass of *BankAccount* to our Person constructor. Our subclass doesn't allow us to withdraw any money, but we have been able to deposit £1000.00.

You may be wondering why we're persisting each transaction into a `val`. This is because we've designed our classes to be immutable and to return a new instance of `BankAccount` each time a method is called. We therefore store the state that `BankAccount` is in at each transaction.
Each value stored in the `val` `cashisa` and `deposited` will remain unchanged for the lifetime of the application.

Let's use polymorphism to change the behaviour of our CashISASavingsAccount deposit method

We're going to change the default implementation of our `deposit` method for our `CashISASavingsAccount` so that it has a limit on how much can be deposited in each transaction, similar to how ISA's work in real life.

Change your implementation of the `deposit` method for your `CashISASavingsAccount` to be the following:

```
  override def deposit(amount: Double): BankAccount = {
    if (amount > depositThreshold) {
      val difference = amount - depositThreshold
      println(s"You can't deposit more than £$depositThreshold.
        Excess: £$difference.")
      new CashISASavingsAccount(accountNumber,
        balance + depositThreshold)
    } else {
      new CashISASavingsAccount(accountNumber,
        balance + amount)
    }
  }
```

You will also need to implement a private property to your `CashISASavingsAccount` class like the following:

```
final class CashISASavingsAccount(accountNumber: String,
                                  balance: Double) extends BankAccount(
                                    accountNumber, balance) {
  private val depositThreshold : Double = 200.00

  ...
}
```

Now re-run your application, you will see that you will receive a new message on your console saying that you can't deposit more than £200.00 per transaction, your current balance should be £200.00.

Commit your code!

Is there a way that we can improve this code? What if we have a very loyal customer who want to be able to deposit more than £200.00 per transaction? We can't currently achieve that with our current implementation.

We can use a feature of Scala called default arguments, this allows us to provide a *constructor* argument to our class where it will provide a default value for a property if we do not explicitly

provide it one.

Let's see this in action, change your implementation to look like the following:

```
final class CashISASavingsAccount(
    accountNumber: String,
    balance: Double,
    private val depositThreshold : Double = 200.00) extends BankAccount(
    accountNumber, balance) {

  override def withdraw(amount: Double): BankAccount = {
    println(s"You can't withdraw yet, your money is locked in for 3 years!!!
      And... we've reduced your APR to 0.2%!")
    this
  }

  override def deposit(amount: Double): BankAccount = {
    if (amount > depositThreshold) {
      val difference = amount - depositThreshold
      println(s"You can't deposit more than £$depositThreshold.
        Excess: £$difference.")
      new CashISASavingsAccount(accountNumber, balance + depositThreshold)
    } else {
      new CashISASavingsAccount(accountNumber, balance + amount)
    }
  }

}
```

Further reading

Scala functions - named arguments and default arguments | alvinalexander.com

As you can see, we have changed our `CashISASavingsAccount` constructor to accept an argument `depositThreshold`. We have set a *private* modifier on the property so that it cannot be accessed from outside of the class. We have also chosen to set this as a *val* so that the value £200.00 cannot be changed once the class has been instantiated.

Now, re-run your code and see what happens!

As you can see, we are still limited to having a deposit threshold of £200.00. Let's change how we instantiate our class to override the default argument. `val cashisa = new CashISASavingsAccount("45676", 0.0, 1000.00)`

Re-run your code, you should now be able to deposit £1000.00 per transaction now since you're a very loyal customer!

Let's instantiate another `BankAccount` so that it uses the default argument.

```
val normalAccount = new CashISASavingsAccount("12334", 100.00)
val loyalAccountDeposited = normalAccount.deposit(300.00)
```

You will also want to implement a new `Person` class that uses our new *BankAccount*.

```
val loyal = new Person("Loyal customer", 22, loyalAccountDeposited)
println(loyal.speak())
```

Now re-run your application and see what happens.
As you can see we have deposited £300.00 which is £100.00 over the limit we are allowed for a normal account.

Referring to the examples above, can you see anything that you would change?

1. Do you think it is a good idea to allow a customer to withdraw from a `CashISASavingsAccount` when this is restricted?
2. How do you think we could change the design of our classes so that you don't have to implement the `withdraw` method for a savings account?

Recap

We have demonstrated how to use access modifiers in order to add a deposit limit to accounts. We have implemented string interpolation in order to log out how much is in each account.

We have also covered:

- Access modifiers
- Secondary constructors
- String interpolation
- Named arguments / default arguments

Resources

- How to create multiple class constructors in Scala | alvinalexander.com
- Scala functions - named arguments and default arguments | alvinalexander.com