Programmation multiparadigme avec Scala
Première étape
De Java à Scala avec des objets *fonctionnels*

Jacques Noyé

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

10 janvier 2017

1 Objectif

L'objectif de ce TP est essentiellement de s'habituer à la syntaxe de Scala tout en travaillant avec des objets fonctionnels. Dans un deuxième temps, on verra aussi cette particularité de Scala qui permet à l'utilisateur de définir ses propres conversions de type.

2 Point de départ

Hypothèse : vous disposez d'une version opérationnelle d'eclipse avec un plugin Scala (voir le forum des nouvelles).

Récupérez sur Campus le projet Java `ComplexJava`. Importez-le en tant que projet Java et vérifiez qu'il fonctionne correctement : l'exécution du point d'entrée `Test.main` doit retourner un score de 36 tests réussis sur 36.

Ce projet correspond à une implémentation des nombres complexes en Java [1]. Cette implémentation utilise une conception descendante partant d'une définition sous forme d'interfaces Java des différents ensembles de *services* qu'il est possible d'associer à une modélisation par objets des nombres complexes (voir figure 2).

3 Implémentation en Scala

Dans un premier temps, nous allons simplement réécrire cette première version en Scala dans un nouveau projet, Scala cette fois, `ComplexV1`, sans en modifier la structure.

On peut écrire en Scala dans un style très proche de Java, mais l'idée va être plutôt d'essayer de profiter des facilités syntaxiques de Scala pour écrire de manière concise en réduisant au maximum le nombre de points-virgules, de crochets, de parenthèses et d'accesseurs inutiles [2].

3.1 Implémentation de `Angle`, `Comparison` et `Conversion`

Commencez par réécrire les classes `Angle`, `Comparison` et `Conversion` en Scala, sachant que les classes `Comparison` et `Conversion` n'étant constituées que de méthodes statiques, on utilisera une classe singleton (mot-clef `object`) en Scala.

1. Il est dramatiquement incomplet. Voyez-vous pourquoi ?
2. Ce dernier point est un point sémantique sur lequel on reviendra en cours.

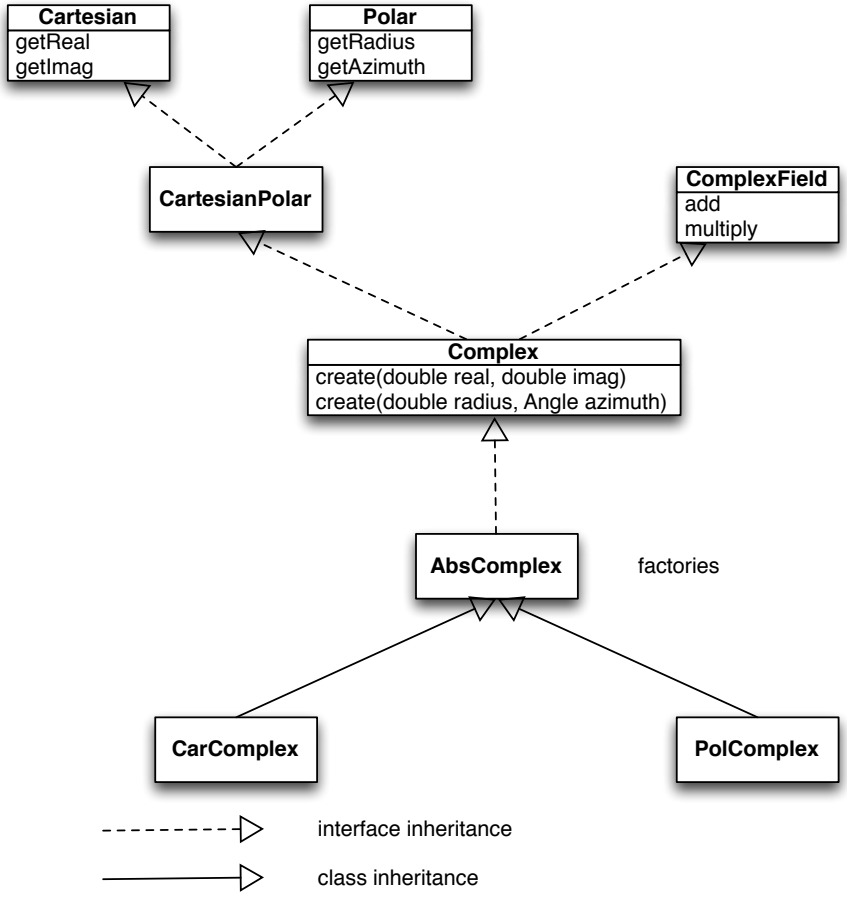

FIGURE 1 – Conception descendante en Java

Pour ce qui est des constantes et méthodes mathématiques usuelles, il est possible de garder les versions Java (de `java.lang.Math`) ou de changer la majuscule de `Math` pour une minuscule et d'utiliser les versions Scala (de `scala.math`). `PI` devient alors `Pi`.

Notez qu'il est possible d'importer un ensemble de noms d'un même paquetage à l'aide d'une notation ensembliste. Par exemple :

```
import Math.{PI, floor} // ou import math.{Pi, floor}
```

Rappelez-vous aussi que vous avez plus de liberté pour nommer vos méthodes.

Test à l'aide de l'interpréteur/REPL Pour testez ces premiers éléments, essayez l'interpréteur Scala : sélectionnez le paquetage (normalement `complex`) dans lequel vous avez défini vos méthodes et, à l'aide d'un clic contextuel, sélectionnez `Scala->Create Scala interpreter in complex`[3].

La commande `:help` vous donne accès à l'ensemble des commandes disponibles.

Notez que si vous modifiez des fichiers, ces modifications ne seront (malheureusement) pas répercutées dans l'interpréteur et vice-versa.

Test à l'aide d'une feuille de travail Alternativement, vous pouvez créer une feuille de travail (*Scala Worksheet*) en vous positionnant soit sur le projet, soit sur le paquetage. Dans le premier cas, il est utile d'importer toutes les définitions visibles du paquetage `complex` (`import complex._`) afin de ne pas avoir à qualifier tous les noms issus du paquetage avec le nom du paquetage.

Il vous suffit ensuite d'ajouter vos tests dans l'objet créé dans la feuille de travail, un test par ligne. À chaque modification et sauvegarde de la feuille de travail, tous les tests sont exécutés et les résultats ajoutés à la feuille de travail sous forme de commentaires.

3.2 Implémentation des interfaces

La construction `interface` de Java se traduit en Scala en une construction `trait` qui, de la même manière, inclut des signatures de méthodes.

Lorsqu'en Java on utilise pour une propriété p un accesseur en lecture (*getter*) `get`P, il suffit d'utiliser en Scala, d'après le principe d'accès universel, une méthode p (sans paramètre).

L'héritage des traits est très proche de l'héritage des classes. Syntaxiquement, le même mot-clef `extends` est utilisé dans les deux cas.

Lors de l'héritage de plusieurs traits[4], les traits ne sont pas séparés par des virgules mais par le mot-clef `with`.

3.3 Implémentation des classes restantes

Terminez par l'implémentation des classes `AbsComplex`, `CarComplex` et `PolComplex`.

Pour la classe `AbsComplex`, la méthode `equals` qui prend en Java un argument de type `Object`, la racine des types des objets, prend en Scala un argument de type `Any`.

Notez aussi les variations de syntaxe pour le test de type et la coercition en Java et Scala :

	Java	Scala
Test de type	o `instanceof` T	o`.isInstanceOf[`T`]`
Coercition	`(` T `)` o	o`.asInstanceOf[`T`]`

Vous pouvez de même que précédemment tester votre code à l'aide de l'interpréteur. Le fichier `TestV1.scala`, disponible sur Campus, vous fournit aussi une traduction de la classe `Test` du projet Java.

Notez l'utilisation de l'opérateur `==` pour l'égalité *naturelle* en Scala (`equals` en Java).

3. Notez la manière dont toutes les définitions du paquetage sont importées (utilisation d'un souligné plutôt que d'une étoile comme en Java).

4. Comme en Java, on ne peut hériter que d'une seule classe et il est aussi possible d'hériter d'une classe et de plusieurs traits.

4 Utilisation de conversions *implicites*

Nous allons maintenant consider la coexistence dans un projet Scala `ComplexV2` de deux classes `CarComplexe` et `PolComplex` qui n'implémentent qu'un type de coordonnées et les opérations gérables à l'aide de ces coordonnées :

— des coordonnées cartésiennes, la somme, l'inverse et l'égalité pour `CarComplex` ;

— des coordonnées polaires, le produit, la racine carré et l'égalité pour `PolComplex`.

Les classes `Angle`, `Comparison` et `Conversion` peuvent être réutilisées.

4.1 Implémentation de la somme

Considérons tout d'abord, le cas le plus simple d'une classe `CarComplex` qui ne comporte que des coordonnées cartésiennes et la somme (qui peut être obtenu en adaptant sa définition dans la classe `AbsComplex` de la version précédente en utilisant un argument qui est aussi un complexe en coordonnées cartésiennes, de même pour le nouveau complexe retourné par la méthode) et d'une classe `PolComplex` qui ne comporte que des coordonnées polaires.

Première situation Soit la somme `a + b` d'un complexe en coordonnées cartésiennes `a` et d'un complexe en coordonnées polaires `b` (voir le fichier `TestV2.scala` pour des exemples de valeur).

L'argument `b` n'est pas du bon type. Il doit être converti en un complexe de coordonnées cartésiennes. Ceci peut être réalisé automatiquement en rajoutant une conversion implicite :

```
implicit def pol2Car(pol: PolComplex): CarComplex
```

Rajouter cette conversion (à implémenter à l'aide de la classe `Conversion`) et vérifier que la conversion est effectivement effectuée. Vous pouvez vous inspirer de l'exemple du cours sur les nombres rationnels (le projet Scala `Rationals.zip` sur Campus permet notamment d'expérimenter avec la commande *Toggle Implicits Display*).

Situation inverse Considérons la somme `b + a`. Que se passe-t-il ? Pourquoi est-ce que l'utilisation de l'opérateur `++` (voir `TestV2.scala`) résout la question ? Est-il possible de faire autrement ?

Dernière situation Que se passe-t-il si les deux opérandes sont des complexes en coordonnées polaires ?

4.2 Implémentation des autres opérations

Une fois le cas de la somme compris, vous pouvez :

1. implémenter le cas dual de la multiplication, qui va demander une nouvelle conversion implicite d'un complexe en coordonnées cartésiennes en un complexe en coordonnées polaires ;
2. les opérations d'inversion et de racine carrée ;
3. le test d'égalité, qui pose un nouveau défi `:-)`