Université de Bordeaux
Année 2018-2019

Mastère 2
T.A.P

TD n°8 - Programmation par aspects

▷ Scala est un langage de programmation conçu par Martin Odersky et dévelopé à l'EPFL (http://www.scala-lang.org). Il peut être compilé vers du bytecode Java et s'exécuter au sein de la machine virtuelle Java. Pour compiler un programme Scala, il suffit de :

- Compilation : utiliser le compilateur `scalac` pour transformer un fichier `.scala` en un ensemble de fichiers `.class` :

  ```
  scalac file.scala
  ```

- Execution : exécuter le code à l'aide de la machine virtuelle :

  ```
  scala file
  ```

  La machine virtuelle `scala`, si elle est lancée seule, peut être utilisée comme une REPL pour le langage Scala.

Voici un exemple très simple de programme Scala :

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Chapito les tepos")
  } // This is a comment
}
```

Exercice 1: Implicits in Scala

En Scala, le mot-clé `implicit` permet à ce que des paramètres de fonctions soient déterminés de manière implicite. Concrètement, certaines valeurs peuvent être marquées comme étant `implicit`, et lorsque le programme en a besoin, il recherche parmi ces valeurs laquelle peut être utilisée. Les règles concernant les implicites sont plutôt complexes et sont expliquées dans le livre *Programming in Scala* (http://www.artima.com/pins1ed/implicit-conversions-and-parameters.html). En particulier, une erreur se produit lorsque plus de deux implicites différents peuvent convenir lors de la phase de recherche.

Une application de cette construction consiste à ajouter des conversions implicites de type dans le langage. Considérons pour l'instant une classe `Complex` très simple permettant

de représenter des nombres complexes [1], fournie avec un objet compagnon :

```
class Complex(val real : Double, val imag : Double) {
  def +(that: Complex) =
    new Complex(this.real + that.real, this.imag + that.imag)
  def -(that: Complex) =
    new Complex(this.real - that.real, this.imag - that.imag)
  override def toString =
    real + "␣+␣" + imag + "i"
}
object Complex {
  def I = new Complex(0, 1)
  implicit def Double2Complex(value : Double) =
      new Complex(value, 0.0)
}
```

Remarquer que l'objet compagnon contient une fonction permettant de construire un objet de type `Complex` simplement à partir d'une valeur de type `Double`.

1. Construire une valeur de type `Complex` à partir d'une valeur de type `Double`, sans faire appel à la fonction `Double2Complex`.
2. Vérifier qu'il est possible d'additioner ou de soustraire un `Complex` avec un `Double`.

Une utilisation des implicites permet d'étendre le comportement d'objets existant sans avoir à modifier ni leur code, ni leur type. Dans l'exemple suivant, on étend la classe `Complex` en lui ajoutant une méthode de comparaison.

```
class OrderedComplex (override val real : Double, override val imag : Double)
  extends Complex(real, imag) with Ordered[OrderedComplex] {
    def compare(that : OrderedComplex) : Int = {
      if (this.real < that.real) -1 else 1 }
}
```

3. Écrire une méthode implicite `Complex2OrderedComplex` qui transforme un `Complex` en sa classe dérivée.
4. Vérifier que, après avoir inclus la méthode précédente, il devient possible de directement comparer des valeurs de type `Complex`.

▷ AspectJ (`http://www.eclipse.org/aspectj`) est une extension du langage de programmation Java permettant de faire de la programmation par aspects. Son utilisation est facilitée depuis l'interieur de l'environnement Eclipse, à travers les AJDT (AspectJ Development Tools, `http://eclipse.org/ajdt`).
L'adresse d'installation des AJDT pour Eclipse Neon est `http://download.eclipse.org/tools/ajdt/46/dev/update`.

1. Exemple tiré de `tomjefferys.blogspot.fr/2011/11/implicit-conversions-in-scala`

Exercice 2: Hello, aspect

Considérons le fichier d'exemple (très simple) Java suivant, que nous allons instrumenter à l'aide d'aspects en AspectJ :

```
class Hello {
    public void print_int(int i) {
        System.out.println("Just␣an␣integer␣:␣" + i);
    }

    public void print_nothing() {
        System.out.println("That's␣nothing");
    }

    public static void main(String [] args) {
        System.out.println("Howdy␣?");
        Hello h = new Hello();
        h.print_nothing();
        h.print_int(42);
        h.print_nothing();
        System.out.println("Hasta␣la␣vista␣...");
    }
}
```

▷ Un aspect se présente sous la forme d'une classe Java utilisant le mot-clé `aspect`, et contenant en plus des méthodes et attributs usuels :

— des *pointcuts*, définissant des points du code à partir desquels il est possible d'insérer les aspects, sous la forme d'une expression booléenne :

```
pointcut tournevis(): execution(* Tournevis.*(..)) || call(void *.main(..));
```

— des *advices*, définissant le code à insérer avant (`before`), après (`after`) voire pendant (`around`) les appels réalisés aux pointcuts.

```
before() : tournevis() {
        System.out.println("Advice␣from␣:␣" +
          thisJoinPoint.getSignature().getName()); }}
```

Ainsi, l'aspect suivant permet de rajouter du code s'exécutant avant l'exécution de toutes méthodes de la classe `Tournevis`.

```
aspect PinceCoupante {
    pointcut tournevis(): execution(* Tournevis.*(..));
    before() : tournevis() {
        System.out.println("Advice␣from␣:␣" +
          thisJoinPoint.getSignature().getName());}
}
```

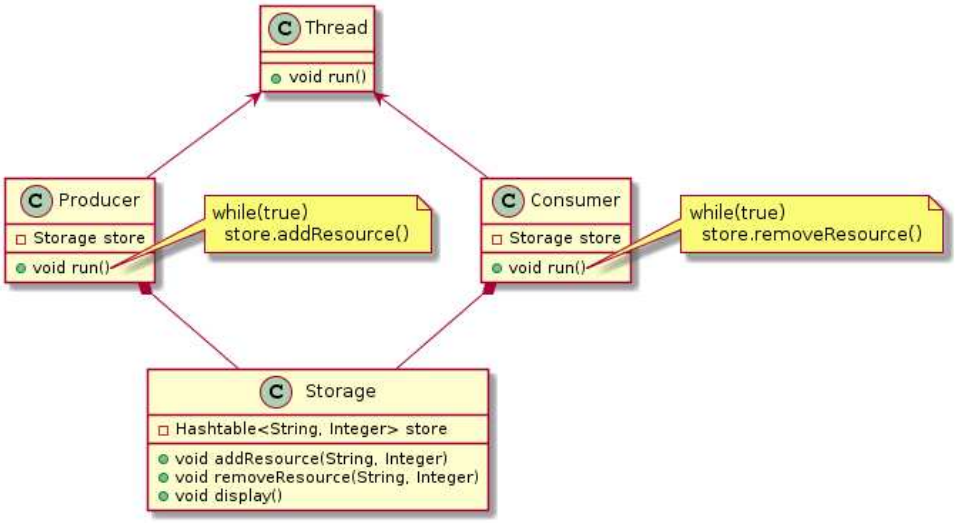
Une courte liste des éléments de langage appartenant à AspectJ se trouve à l'adresse `http://eclipse.org/aspectj/doc/released/progguide/quick.html`.

1. Ajouter un premier aspect qui, pour toutes les méthodes de la classe `Hello`, affiche un message avant l'appel de la méthode et un message après cet appel.
2. Créer un second aspect qui ne capture que la méthode `print_int`, en s'inspirant de la syntaxe suivante :

```
pointcut justints(int x) : execution(* Hello.*(int)) && args(x);
before(int x) : justints(x) { /* ... */ }
```

Exercice 3: Stockage de myrtilles

Plusieurs threads doivent partager une ressource commune, dans un système producteur/-consommateur. Dans notre exemple, la ressource commune est une classe nommée `Storage` possédant deux méthodes d'ajout (`addResource`) et de retrait (`removeResource`). Un thread `Producer` s'occupe de remplir le `Storage`, tandis que le `Consumer` s'occupe de le vider. Le diagramme de classes suivant décrit l'architecture du projet :



⚠ Sauf mention explicite, il est demandé de ne pas modifier le code existant.

1. Créer un nouveau projet AspectJ sous Eclipse, dans lequel vous inclurez les fichiers source suivants :

   `Consumer.java`, `EmptyStorageException.java`, `LockedStorageException.java`, `Main.java`, `Producer.java`, `Storage.java`

   Rappel : pour importer des fichiers dans un projet sous Eclipse, suivre le sinueux chemin ⟨*Project name*⟩ → src → Import → General → File System.

   Vérifier que l'exécution de la classe `Main` s'exécute sans problèmes en n'affichant qu'une seule ligne de texte.

Tel quel, ce code n'est pas particulièrement pratique à manipuler : il n'effectue aucun affichage permettant de savoir si le code s'exécute. Plutôt que de modifier le code existant (a fortiori uniquement pour des fins de débuggage), nous allons écrire un *aspect* permettant d'afficher des messages à l'écran.

2. Ajouter un aspect `Log.aj` dans le code affichant un message lors de l'entrée et de la sortie des méthodes de la classe `Storage`, et toute autre méthode considérée comme intéressante pour cette question.

Remarque : Il n'existe pas de mécanisme simple en AspectJ permettant d'activer ou de désactiver des aspects. Pour faire simple, il est recommandé de commencer chacun de vos aspects avec une variable `ENABLED` :

```
final static boolean ENABLED = true;
pointcut marteau() : if (PinceCoupante.ENABLED) && // ...
```

Les aspects peuvent accéder à certaines informations du contexte appelant, en utilisant une variable nommée `thisJoinPoint`. Par exemple l'objet appelant peut être obtenu par

`thisJoinPoint.getTarget()` (cf. `https://eclipse.org/aspectj/doc/released/runtime-api/org/aspectj/lang/JoinPoint.html` pour les méthodes appelables sur cet objet).

3. Étendre l'aspect précédent en ajoutant un appel à la méthode `Storage.display()` lors de toute modification d'un objet `Storage`. Faire de manière à ce que les deux aspects soient activables indépendamment l'un de l'autre.

Exercice 4: Race conditions

Un problème important lorsque l'on manipule des threads provient du partage des ressources, partage qui peut mener à des situations de concurrence (*race condition*). Ici, l'accès à la ressource `Storage` est fait de manière concurrente par plusieurs threads. Nous proposons de sécuriser ce mécanisme à travers un aspect, en ajoutant un mutex Java pour chaque objet de la classe `Storage` (sous la forme d'un `Lock` de la classe `ReentrantLock`) [2].

1. Combien d'aspects sont nécessaires dans ce cas particulier ?

   Remarque : regarder la rubrique « Per-object aspects » de `https://eclipse.org/aspectj/doc/released/progguide/semantics-aspects.html`

   Proposer un pointcut ainsi qu'un ou plusieurs advices permettant d'implémenter un aspect créant un objet de type `ReentrantLock` pour chaque objet de type `Storage`, et utilisant les méthodes `lock()` et `unlock()` pour encadrer les appels aux méthodes de la classe `Storage`.

▷ L'utilisation du pointcut `around` permet de transmettre les arguments passés fonction, tout en ajoutant du code avant et après l'appel à cette fonction.

```
public aspect CaptainAgeAspect {
    pointcut setAge(Integer i): call(* setAge(..)) && args(i);

    Object around(Integer i): setAge(i) {
        System.out.println("Before␣that,␣he␣was␣" + i + "years␣old.");
        Integer newi = (Integer) proceed(i*2);
        System.out.println("After,␣he␣got␣older␣:␣" + newi + "years.");
        return newi;
    }}
```

L'utilisation (facultative) du type `Object` permet même de récupérer l'appel à la méthode avant de le retransmettre.

Dans l'exemple précédent, il est possible que les demandes d'entrée en section critique restent en attente. En effet, l'appel à la méthode `lock()` est bloquant pour le thread courant, sans lever d'exception. Raffinons ce comportement, en notifiant par une exception au thread appelant qu'il a dû passer son tour [3].

2. Dans un exemple aussi simple, il est possible de résoudre le problème simplement en utilisant des méthodes `synchronized`. Ici, on propose d'implémenter le mécanisme à la main afin de pouvoir détecter les accès concurrents.

3. Les `ReentrantLock` peuvent gérer une partie de leur scheduling par eux-même, pour éviter de laisser certains threads patienter trop longtemps.

2. Proposer une manière de faire pour lever une exception `LockedStorageException` lorsqu'un appel à `lock()` est fait alors que le Mutex est possédé par un autre thread.

Les implémentations des classes comme `Storage` contiennent déjà les exceptions que l'on vient de rajouter. Dans l'esprit de la programmation par aspect, il serait bon de pouvoir les ajouter directement à travers l'aspect que nous venons d'écrire.

3. Est-il possible, en utilisant AspectJ, d'écrire un aspect faisant qu'une fonction qui initialement ne levait pas d'exception en lève une après application ?
4. Est-il possible, en utilisant AspectJ, de modifier le prototype d'une fonction existante ?

   Remarque : La FAQ d'AspectJ (à l'adresse `http://www.eclipse.org/aspectj/doc/released/faq.php`) peut aider à répondre à ces questions.

Exercice 5: Aspects réutilisables

Il est dommage de ne pouvoir appliquer un tel mécanisme qu'à la classe `Storage`, alors qu'en fait il est naturel de vouloir spécifier des familles de méthodes qui doivent s'exécuter sans concurrence.

1. Inclure dans votre code[4] les fichiers source suivants :
   `Condition.java`, `CoordinationAction.java`, `Exclusion.java`, `Method.java`, `MethodState.java`, `Mutex.java`, `Selfex.java`, `TimeoutException.java`, ainsi que le fichier `Coordinator.aj`

Le fichier `Coordinator.aj` définit un pointcut générique `synchronizationPoint`, et plusieurs advice dépendant de ce pointcut. Il permet de mettre automatiquement en place un système de verrous permettant qu'un ensemble de méthodes ne s'exécutent pas en même temps. Il définit deux classes :

— un `Selfex` correspond à une méthode s'exécutant sans interruption (à la manière de `synchronized`) ;

— un `Mutex` (dans cet aspect) correspond à une liste de méthodes ne devant pas s'exécuter en même temps.

Il est possible d'interagir avec cet aspect simplement avec les méthodes `addMutex` et `addSelfex`, qui prennent en paramètre respectivement un nom de méthode (sous forme de chaîne de caractères) et un tableau de noms de méthodes.

2. Écrire un aspect étendant `Coordinator`, définissant un pointcut correct, afin de faire que les méthodes `addResource` et `removeResource` ne puissent pas être exécutées en même temps par deux threads différents.
3. Quels problèmes de maintenabilité la programmation avec AspectJ pose t'elle ?

4. Exemple tiré de la documentation d'AspectJ 1.7