Maitrise d'Informatique
Module POD

Programmation Objet et Distribution
Documents autorisés

Université Pierre et Marie Curie
Année 1998–1999
Durée 3 heures

Examen du 29 janvier 1999

1 Classes paramétrées en Java

Le but de cet exercice est de réfléchir à l'implantation d'un mécanisme de classes paramétrées en Java. On appliquera cette technique à la construction d'ensembles homogènes. On étend la syntaxe de Java on ajoutant un paramètre de type à la manière de C++.

```
class StoreSomething<A> {
     A something;

     StoreSomething(A something) {
         this.something = something;
     }

     void set(A something) {
         this.something = something;
     }

     A get() {
         return something;
     }
}
```

où la classe `StoreSomething` a un paramètre de type `<A>`. Son utilisation est par exemple :

```
StoreSomething<String> a = new StoreSomething("I'm a string!");
StoreSomething<int> b = new StoreSomething(17+4);

a.set("New string");
b.set(9);

String s = a.get();
int i = b.get();
```

L'héritage d'une classe paramétrée peut être non paramétré :

```
class StoreString extends StoreSomething<String> {
    StoreString(String something) {
        super(something);
    }
}
```

ce qui donne une sous-classe spécialisée d'une classe paramétrée, ou bien paramétré :

```
class StoreSomething2<A> extends StoreSomething<A> {

          StoreSomething2(A something) {
              super(something);
          }

          void print() {
              System.out.println(""+something);
          }
      }
```

qui donne une sous-classe paramétrée d'une classe paramétrée.

Il y a 2 manières de compiler cette extension en pur Java. La première traduit chaque classe paramétrée spécialisée (par un type) en une nouvelle classe : cela favorise l'efficacité du code qui sera alors plus gros, la deuxième traduit chaque classe paramétrée en une classe où le paramètre de type devient `Object` et efffectue des coercions de type à chaque appel spécialisé : cela produit un code petit mais ajoute de nombreux tests dynamiques de types.

1. Traduisez, à la main, la définition de la classe `StoreSomething<A>` en deux classes `StoreSomething_int` et `StoreSomething_String` correspondant aux 2 spécialisations de cette classe dans le programme principal. Modifier en conséquence le programme principal.
2. Traduisez, à la main, la définition de la classe `StoreSomething<A>` en une classe `StoreSomething` où le type `<A>` est remplacé par le type `Object`. Modifier en conséquence le programme principal en ajoutant les coercions de types nécessaires. Attention vous ne pouvez pas coercer directement un `Object` en `int`.
3. Indiquez le comportement de chaque traduction pour les 2 cas d'héritage donnés en exemple. Vous utiliserez d'autre part le même programme principal où les déclarations `StoreSomething` deviennent des `StoreSomething2`.
4. Décrivez comment automatiser vos traductions manuelles par un programme engendrant les classes Java correspondantes.
5. Ecrire une classe `Liste<A>` ou une hiérarchie de classes pour des listes homogènes.

2 Moniteurs en O'CAML

On s'intéresse ici à implanter en O'CAML un mécanisme de moniteurs à la Java. C'est à dire à l'implantation de méthodes *synchronized* dans des déclarations de classes.

1. Définir une classe `monitor` ayant comme champs de données un `Mutex` m et une `Condition` c, et comme méthodes définies les méthodes `lock` qui verouille et `unlock` qui libère le `Mutex` m et les méthodes `wait` qui libère le mutex sur la condition ainsi que les méthodes `notify` et `notifyAll` qui indique un changement de condition aux autres threads. Le constructeur de cette classe créera les valeurs du mutex et de la condition.
2. On rappelle le code de la classe `shop` vue en cours :

```
let m = Mutex.create();;
let c = Condition.create();;

class shop n =
object(self)
  val  mutable size =n;
  val  mutable buffer = ([||] : product array)
  val  mutable ip = 0
  val  mutable ic = 0

  initializer buffer<-Array.create 12 (new product "empty")

   method display1 () = ...
   method display2 () = ...

  method put  = function p ->
    Mutex.lock m;
    while (ip-ic+1 > Array.length(buffer)) do Condition.wait c m done;
    buffer.(ip mod size) <- p;
    self#display1();
    ip <- ip+1;
    Condition.signal c;
    Mutex.unlock m
```

```
  method get  = function () ->
    Mutex.lock m;
    while(ip == ic) do Condition.wait c m done;
    self#display2();
    let r = buffer.(ic mod size) in
      ic<- ic+1;
      Condition.signal c;
      Mutex.unlock m;
      r
end;;
```

Reformulez la classe `shop` comme une sous-classe de `monitor` en utilisant uniquement les méthodes héritées pour la synchronisation de `put` et `get`.

3. On cherche à simuler le fonctionnement du mot clé *synchronized* de Java en O'CAML. Indiquez les modifications à apporter en début et en fin de code des méthodes `get` et `put` pour simuler ce comportement. Ecrivez une macro `synchronized`, au sens de `cpp`, qui englobe l'expression de la méthode en ajoutant le code nécessaire.

3 Client-serveur pour la synchronisation de date

Le but de cet exercice est d'implanter un service de mise à jour des dates, à la manière de `rdate` (RFC 868). Le serveur sera écrit en O'CAML et les clients en Java. Voici le schéma du service de date :

```
S: écoute sur le  port 37.

C: connexion sur le port 37.

S: envoi du temps comme un nombre flottant en format texte.

C: réception du temps.

C: fermeture de la connexion.

S: fermeture de la connexion.
```

Le temps de ce protocole est compté en secondes écoulées depuis le premier janvier 1900 (GMT). En O'CAML la fonction `Unix.time` retourne le temps écoulé depuis le 1er janvier 1970 en secondes. En Java on utilisera la classe `java.util.Date` dont le constructeur sans argument initialise l'instance avec la date courante. La méthode `long getTime()` récupère une date et la méthode `void setTime(long)` la modifie. Le *long* correspond au nombre de millisecondes écoulées depuis le 1er janvier 1970. On appellera `decal` la constante du nombre de secondes entre 1/1/1900 et 1/1/1970. On utilisera la méthode statique (classe `long`) `public static long valueOf(String s) throws NumberFormatException` pour convertir une chaîne en *long*.

1. Ecrire le serveur O'CAML répondant à ce protocole. Le serveur doit pouvoir répondre à plusieurs requêtes simultanées.
2. Ecrire un client Java simple, demandant une date et l'affichant. Il n'est pas demandé d'écrire la fonction de conversion de secondes en date.
3. Modifier le client de telle manière qu'il envoie une requête via une *thread*.

On cherche à améliorer ce protocole client-serveur pour contre balancer les effets dus à l'encombrement du réseau. Pour cela le client lance 3 requêtes distinctes en conservant la date de l'émission. Quand le serveur répond, le client construit 1 triplet : (date émission, date envoyée du serveur, date de réception). Le client détermine ensuite selon les 3 triplets quelle est la différence de date entre lui et le serveur et se met à jour.

4. Proposer une solution pour la mise à jour de l'heure en tenant compte des 3 triplets.
5. Implantez votre solution en modifiant le client en conséquence.