

Les références constantes

Philippe PRADOS

pp@philippe.prados.name



*Préservez l'environnement,
n'imprimez pas ce document*

TABLE DES MATIERES

1.	Pourquoi contrôler l'usage des références ?	3
2.	Les différentes solutions	3
2.1	Instances immuables	3
2.2	Retourner la valeur d'un attribut.....	4
2.3	Références constantes.....	4
2.3.1	Identifier les méthodes constantes	4
2.3.2	Proposer une référence constante	5
3.	Conclusion	6

Avant propos

Il n'est pas possible, avec java, de contrôler l'usage qu'une classe fera d'un pointeur. Le C ANSI et le C++ proposent des références constantes. Nous allons voir comment offrir ce concept avec Java.

1. POURQUOI CONTROLER L'USAGE DES REFERENCES ?

Lors de la rédaction d'un framework Java, il faut se prémunir d'une utilisation erronée des instances. En effet, le framework doit pouvoir évoluer sans remettre en cause les développements précédents. Java n'offre pas d'outils syntaxiques pour limiter l'usage qu'un client du framework fera d'une référence. Par exemple, un accesseur retourne une référence à un attribut interne. Le pointeur sur l'attribut peut être mal utilisé par le client. L'instance retournée pourra être modifiée à l'insu de l'instance propriétaire.

```
class Framework
{ private MonObjet attr_=new MonObjet();
  public MonObjet getAttr()
  { return attr_;
  }
  public void setAttr(MonObjet x)
  { attr_=x;
  }
}
//...
class ClientFramework
{ MonObjet ref_;
  //...
  void method(Framework f)
  { setRef(f.getAttr()); // Oupps !
  }
  void setRef(MonObjet x)
  { ref_=x;
  }
  void update()
  { ref_.setValue(100); // Glups !
  }
}
```

La classe `Framework` possède une agrégation privée `attr_` de type `MonObjet`. Elle propose un accesseur pour contrôler les manipulations à son attribut.

La classe `ClientFramework` garde par erreur le pointeur retourné par l'accesseur (`Oupps !`). Par la suite, elle peut apporter des modifications à l'objet pointé par `attr_` (`Glups !`), à l'insu de l'instance `Framework`. Cela viole le principe de l'encapsulation. Un client du `Framework` ne doit pas apporter de modification à l'instance `MonObjet`. L'accesseur est presque équivalent à rendre l'attribut `attr_` publique. Il interdit la modification de la valeur de `attr_` mais n'interdit pas de modifier l'instance pointée.

Pour la première version de la classe `Framework`, cela fonctionne. Mais, demain, lorsque la classe `Framework` évoluera, le code du client peut ne plus fonctionner.

```
class Framework
{ private MonObjet attr_=new MonObjet();
  private int total_;
  public MonObjet getAttr()
  { return attr_;
  }
  public void setAttr(MonObjet x)
  { attr_=x;
    total_=x.getValue()*2;
  }
}
```

Avec la nouvelle version de la classe, l'attribut `total_` n'est pas correctement valorisé si le client appelle la méthode `setValue()` sur l'instance pointée par `attr_`. L'erreur du client apparaîtra à ce moment et sera très difficile à localiser.

2. LES DIFFERENTES SOLUTIONS

Pour limiter ces erreurs, il y a plusieurs stratégies possibles.

- Utiliser des instances immuables.
- Retourner la *valeur* d'un attribut.
- Retourner une référence constante.

2.1 *Instances immuables*

Une approche, utilisée par la classe `String` de java (idem en Smalltalk), consiste à créer des objets immuables. Une instance de la classe `String` n'évolue plus lors de sa durée de vie. Aucune méthode de `String` ne modifie l'instance courante. Heureusement ! Sinon, il y aurait de nombreuses erreurs difficiles à trouvées dans les programmes Java. Pratiquement toutes les classes gardent des pointeurs sur des `Strings` reçus en paramètre.

```
class Personne
{ String nom_;
  String prenom_;
  public Personne(String nom,String prenom)
  { nom_ =nom;
    prenom_=prenom;
  }
}
```

Si la classe `String` n'était pas immuable, ce code serait erroné. En effet, une instance `Personne` pourrait, par exemple, convertir en majuscule le `nom_`, cela modifierait le paramètre `nom` qui a de très forte chance d'être utilisé ailleurs.

Le ramasse-miettes est un très bon assistant des instances immuables. Il va permettre d'éviter de dupliquer les chaînes de caractères et fera le ménage lorsque cela sera nécessaire. Si une instance `String` était modifiable, il faudrait rédiger un code comme celui là :

```
class Personne
{ String nom_;
  String prenom_;
  public Personne(String nom,String prenom)
  { nom_ =new String(nom);
    prenom_=new String(prenom);
  }
}
```

Les instances immuables sont très intéressantes pour certains objets, mais impraticables avec l'ensemble des classes. Il n'est pas raisonnable de figer les valeurs d'une instance lors de sa construction, et de ne plus y toucher par la suite. Sinon, chaque méthode désirant modifier une instance devrait créer une nouvelle instance en recopiant les attributs de l'instance courante et en modifiant certains.

```
class Personne
{ String nom_;
  String prenom_;
  //...
  Personne setNom(String nom)
  { return new Personne(nom,prenom_);
  }
}
```

C'est ce que fait la classe `String` de Java. La méthode `toUpperCase()` retourne une nouvelle instance `String`, mais ne modifie pas l'instance courante.

Dans les situations où une instance ne peut pas être immuable, il faut utiliser une des deux autres approches suivantes.

- Retourner la *valeur* d'un attribut.
- Retourner une référence constante.

2.2 Retourner la valeur d'un attribut

Pour empêcher un client du framework de modifier un attribut interne, il est parfois possible de retourner une copie de l'attribut.

```
class Framework
{ private MonObjet attr_;
  public MonObjet getAttr()
  { return attr_.clone();
  }
}
```

Le client ne reçoit plus un pointeur sur l'attribut, mais un pointeur sur une copie de celui-ci. Il peut faire ce qu'il désire de la copie, cela n'impacte pas le framework. Retourner une copie de son attribut est équivalent à retourner la valeur de l'attribut. C'est ce qu'il se passe lorsque l'attribut est un type primitif. Retourner un entier permet d'obtenir une copie de la valeur de l'entier.

Copier un objet peut être très coûteux en temps et en mémoire. En effet, si `MonObjet` est un arbre complexe et très profond, une copie complète doit être faite pour garantir que le client n'ira pas modifier l'attribut. Cette copie ne servira peut-être à rien, car si le client ne fait que consulter l'attribut, il aurait été possible de lui donner directement l'accès. Pour éviter les copies, il faut retourner une référence constante.

2.3 Références constantes

On constate qu'il existe deux types de méthodes. Les méthodes de consultation et les méthodes de modifications. En C++, on appelle les méthodes de consultations des méthodes constantes.

2.3.1 Identifier les méthodes constantes

Il faut identifier les méthodes constantes des autres méthodes. Une méthode constante ne doit pas modifier l'instance courante et ne doit pas modifier les instances en *agrégation*. Par exemple, une méthode de la classe `Framework` qui appelle la méthode `setValue()`, via `attr_`, n'est pas une méthode constante.

```
class Framework
{ private MonObjet attr_;
  //...
  public void methode()
```

```

    { attr_.setValue(50);
  }
}

```

L'instance courante n'est pas modifiée (l'attribut `attr_` n'a pas bougé), mais la valeur de l'instance pointée par `attr_` est modifiée. `methode()` n'est pas une méthode constante. Il ne doit y avoir aucun effet de bord sur l'instance.

La situation est différente si la référence représente une *relation*. En effet, modifier un objet en relation peut être effectué dans une méthode constante. Cela ne modifie pas l'instance courante.

```

class Framework
{ private MonObjet attr_;
  private Relation rel_;
  //...
  public void methodeConst()
  { rel_.setData("hello");
  }
}

```

`methodeConst()` ne modifie pas, sémantiquement, l'instance `Framework`. Elle modifie un objet en relation. La méthode est constante.

2.3.2 Proposer une référence constante

Une fois identifier les méthodes constantes des méthodes non constantes, comment limiter l'accès aux seules méthodes constantes pour le client ?

Le C++ possède la notion de référence constante. Les références peuvent être qualifiées pour n'autoriser que l'appel aux méthodes constantes. Pour proposer cela avec Java, nous allons utiliser les interfaces. Pour chaque classe du framework, nous proposons une interface regroupant les seules méthodes constantes. Les classes vont implémenter ces interfaces.

```

interface const_MonObjet
{ int getValue();
}

class MonObjet implements const_MonObjet
{ private int _value;
  public int getValue()
  { return _value; }
  public void setValue(int x)
  { _value=x;
  }
}

interface const_Framework
{ const_MonObjet getAttr(); /* const */
}

class Framework implements const_Framework
{ private MonObjet attr_;
  public const_MonObjet getAttr() /* const */
  { return attr_;
  }
  public void setAttr(MonObjet x)
  { attr_=x;
  }
}

```

L'accessor retourne une référence constante, représenté par l'interface `const_MonObjet`. Ainsi, le client peut garder le pointeur s'il le désire, la classe `Framework` est garantie que l'instance référencée par `attr_` n'évoluera pas à son insu.

```

class ClientFramework
{ const_MonObjet ref;
  //...
  void method(Framework f)
  { setRef(f.getAttr());
  }
  void setRef(const_MonObjet x)
  { ref=x;
  }
  void update()
  { ref.setValue(100); // Invalide methode !
  }
}

```

Avec cette solution, nous évitons de devoir copier l'attribut, tous en limitant son accès. Les interfaces débutant par `const_` peuvent être utilisées dans les paramètres afin de prévenir l'appelant de la méthode que le paramètre ne sera pas modifié. Cela sera d'ailleurs vérifié par le compilateur et/ou la VM.

En cas d'héritage, les interfaces constantes devront également hériter entre elles.

```

interface const_Framework
{ public const_MonObjet getAttr(); /* const */
}

class Framework implements const_Framework
{ private MonObjet attr_;
  public const_MonObjet getAttr() /* const */
  { return attr_;
  }
  public void setAttr(MonObjet x)
  { attr_=x;
  }
}

interface const_ExFramework extends const_Framework
{ public const_MonAutreObjet getAutre(); /* const */
}

class ExFramework extends Framework
    implements const_ExFramework
{ private MonAutreObjet autre_;
  public const_MonAutreObjet getAutre() /* const */
  { return autre_;
  }
  public void setAutre(MonAutreObjet x)
  { autre_=x;
  }
}

```

Il y a un arbre d'héritage pour les interfaces constantes, et un arbre d'héritage pour les classes implémentant les interfaces constantes.

3. CONCLUSION

Le mot-clef `const` est réservé par Java. Les concepteurs du langage ont fait une provision pour pouvoir ajouter par la suite ce concept qui améliore grandement la qualité des programmes.

Pour la petite histoire, dans les années 80, un chercheur appelé Stroustrup avait conçu un langage appelé « C avec classe ». Il proposa un petit mot clef « `readonly` » et il démontra sa qualité. Les concepteurs du C au laboratoire Bell ont apprécié cette idée, mais ont préféré l'appeler « `const` ». Le comité de normalisation du C ANSI a retenu cette idée. Par la suite, Bjarne Stroustrup améliora son langage, et le renomma « C++ ».

Il est probable, qu'une version ultérieure de Java proposera nativement ce concept. Il suffira alors de supprimer les interfaces additionnelles, de transformer dans les sources les « `const_xxx` » en « `const xxx` » et le tour est joué.