

Les inners classes

Philippe Prados

pp@philippe.prados.name



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

TABLE DES MATIERES

1.	Les classes de classes.....	3
2.	Les classes de méthodes.....	5
3.	Les classes anonymes.....	6

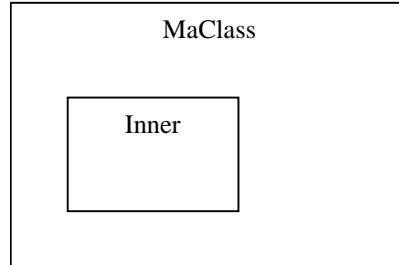
Avant propos

Ce document explique comment le compilateur gère et compile les classes internes.

Le JDK 1.1 propose des inner-classes. Ce sont des classes déclarées dans un contexte particulier, permettant d'accéder à des informations qu'une classe classique ne peut avoir. Cela permet au compilateur d'apporter des optimisations impossibles autrement. Une inner-classe est utilisée dans un contexte bien précis et délimité.

1. LES CLASSES DE CLASSES

La première catégorie d'inner-classes sont les classes déclarées dans une autre classe.

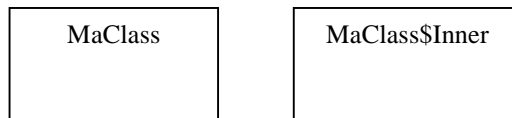


```
class MaClass
{ private int attr_;
  class Inner
  { void f()
    { attr_=5;
    }
  }
  void g()
  { new Inner().f();
  }
}
```

La particularité d'une inner-classe est de pouvoir accéder aux attributs de la classe externe. La méthode `f()` de la classe `MaClass.Inner` peut manipuler l'attribut privé `attr_`. Comment cela fonctionne ?

Les inner-classes sont une extension du langage, mais pas de la machine virtuelle. Une machine virtuelle version 1.0 est capable de manipuler une inner-classe. En fait, le compilateur transforme une classe interne en une classe externe classique. Une classe interne possède une référence vers une instance de la classe externe. Cette référence est initialisée lors de la construction de l'instance interne. Tous les constructeurs de la classe interne possèdent un paramètre caché supplémentaire pour initialiser un attribut vers l'instance externe.

La classe `Inner` est traduite par le compilateur comme ceci :



```
class MaClass$Inner
{ private MaClass this$0;

  MaClass$Inner(MaClass this$0)
  { this.this$0= this$0;
  }
  void f()
  { this$0.attr_=5;
  }
}
```

Pour que cela fonctionne avec une machine virtuelle version 1.0, il faut qu'une classe normale puisse accéder à l'attribut privé `attr_`. Le compilateur transforme, sans le signaler, l'attribut `private` en attribut package-private. Cela est très dangereux pour la sécurité. En effet, un attribut privé qui est utilisé par une inner-classe devient accessible à toutes les classes du package.

La classe externe est modifiée ainsi.

```
class MaClass
{ /*private*/ int attr_; // package private
  void g()
  { new MaClass$Inner(this).f();
  }
}
```

La construction de la classe interne reçoit implicitement `this` en paramètre. On comprend alors pourquoi il n'est pas possible de construire une instance d'une classe interne en dehors d'une méthode de la classe externe. Pour permettre cela, il a fallu ajouter une extension à la syntaxe.

```
MaClass obj=new MaClass();
obj.new MaClass.Inner();
```

Cette syntaxe est traduite par le compilateur en :

```
MaClass obj=new MaClass();
new MaClass$Inner(obj);
```

À l'intérieur d'une méthode de la classe interne, il existe plusieurs instances `this`. En effet, cela peut représenter l'instance courante de la classe interne ou l'instance associée de la classe externe. Pour différencier les deux versions, il faut préfixer la variable `this` du nom de la classe.

```
class MaClass
{ private int attr_;
  class Inner
  { void f()
    { g(MaClass.this);
    }
  }
  void g(MaClass x)
  { ...
  }
}
```

Ce code est traduit par le compilateur en ceci :

```
class MaClass$Inner
{ void f()
  { this$0.g(this$0);
  }
}
```

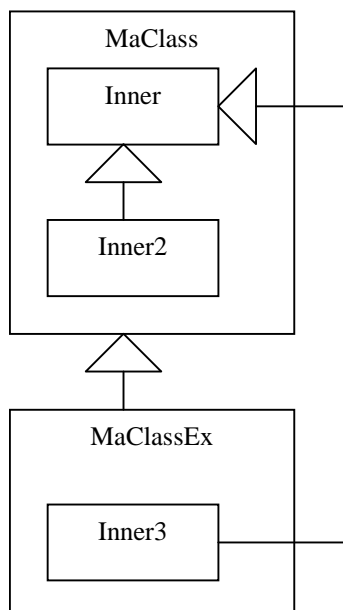
La syntaxe `MaClass.this` permet d'accéder à l'attribut caché `this$0`.

Cette syntaxe peut être nécessaire pour synchroniser les méthodes de la classe interne avec les méthodes de la classe externe.

```
class MaClass
{ private int attr_;
  class Inner
  { void f()
    { synchronized(MaClass.this)
      { ...
      }
    }
  }
}
```

Cela permet de protéger les accès aux attributs privés de la classe externe.

Il est possible d'hériter d'une classe interne en étant une autre classe interne de la même classe ou d'une sous-classe.

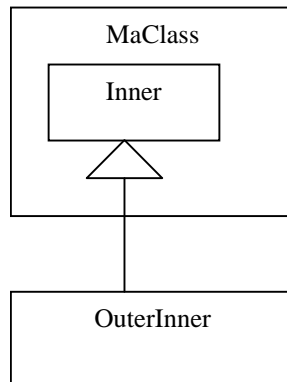


```

class MaClass
{
    class Inner
    {
    }
    class Inner2 extends Inner
    {
    }
}
class MaClassEx extends MaClass
{
    class Inner3 extends Inner
    {
    }
}

```

De même, il est possible de déclarer une classe externe héritant d'une classe interne. Nous avons vu qu'il est nécessaire d'ajouter un paramètre au constructeur de la classe interne. Pour alimenter ce paramètre caché, il faut préfixer l'appel à `super()` avec la référence sur l'instance externe.



```

class OuterInner extends MaClass.Inner
{
    OuterInner(MaClass obj)
    {
        obj.super();
    }
}

```

Pour éviter d'avoir un pointeur vers une instance de la classe externe, il faut déclarer la classe interne avec l'attribut `static`.

```

class MaClass
{
    static class Inner
    {
    }
}

```

Dans ce cas, la classe interne est équivalente à une classe externe, sauf que son nom est préfixé par le nom de la classe externe. La classe interne peut continuer à manipuler les attributs privés.

Les classes internes statiques sont plus rapides que les classes internes classiques. En effet, il n'est pas nécessaire de gérer l'attribut caché. Cela est important pour les optimisations. Par défaut, une classe interne doit être déclarée `static`. Si cela n'est pas possible car la classe interne désire accéder aux attributs de la classe externe, le compilateur signalera le problème. Il faudra alors supprimer l'attribut `static` et recompiler. Dans la situation inverse – utiliser une classe interne non statique alors que ce n'est pas nécessaire – le compilateur ne dira rien.

2. LES CLASSES DE METHODES

Il est possible de déclarer une classe dans une méthode. Cela permet aux instances de cette classe de manipuler les variables locales de la méthode.

```

class MaClass
{
    void g()
    {
        final int i=10;
        class Inner
        {
            void f()
            {
                System.out.println(i);
            }
        }
        new Inner().f();
    }
}

```

La méthode `f()` de la classe `Inner` de la méthode `MaClass.g()` peut manipuler la variable locale `i`. Celle-ci doit être déclarée `final`. Pourquoi ?

Lorsque l'on crée une tâche, on lui associe une pile qui ne sera utilisée que par elle. Il n'y a jamais de conflit d'accès sur la pile. Si la méthode `f()` est appelée dans une autre tâche, il y aurait la possibilité d'avoir simultanément deux accès à la variable `i`. Comme il n'est pas possible de synchroniser l'accès aux variables de la pile, il faut interdire cela. Déclarer une variable `final` permet de garantir qu'elle ne sera jamais modifiée. Cela règle le problème. Deux traitements peuvent consulter la variable, mais aucun ne peut la modifier.

Que se passe-t-il si la méthode `g()` est terminée alors qu'une instance `Inner` est encore disponible ? Où la méthode `Inner.f()` va pouvoir trouver la variable `i` ?

```
class MaClass
{ Object g()
  { final int i=10;
    class Inner
    {
      void f()
      { System.out.println(i);
      }
    }
    return new Inner();
  }
}
```

Dans ce nouveau programme, la fonction `g()` return une instance `Inner`, interne à la fonction. L'appelant de `g()` peut utiliser l'instance interne après l'exécution de `g()`.

Toutes les variables locales utilisées sont recopiées lors de l'appel du constructeur de l'instance. La classe `Inner` est convertie par le compilateur en classe classique comme ceci :

```
class MaClass$1
{ private final int val$i;
  MaClass$1(int val$i)
  { this.val$i =val$i;
  }
  void f()
  { System.out.println(val$i);
  }
}
```

Le compilateur déclare la classe `$1` car elle ne doit pas être confondue avec une classe interne de même nom, mais n'appartenant pas à une fonction. Les classes de fonction ne sont visibles que pour chaque fonction. La classe externe est traduite ainsi :

```
class MaClass
{
  Object g()
  { final int i=10;
    return new Inner(i);
  }
}
```

La classe interne ajoute dans les constructeurs, toutes les variables `final` nécessaires. Les instances internes possèdent une copie des variables locales de la méthode. Il n'y a pas de risque d'incohérence entre les deux copies car elles ne peuvent pas être modifiées.

Si une variable locale à la méthode n'est utilisée que par le constructeur, le compilateur le détecte et ne génère pas d'attribut pour celle-ci. En effet, cette variable n'est plus nécessaire pour les autres méthodes.

3. LES CLASSES ANONYMES

Java propose également la notion de classe anonyme. Une classe anonyme est une classe déclarée là où une variable est nécessaire.

```
class MaClass
{ void f()
  { class MyThread implements Runnable
  { public void run()
    { for (int i=0;i<10;++i)
      System.out.println("Hello");
    }
  };
  new MyThread().start();
}
```

La classe `MyThread` ne sert qu'une seule fois. Son nom importe peu. Pour éviter de devoir déclarer des classes techniques avec un nom, les classes anonymes permettent de simplifier ce code.

```
class MaClass
{ void f()
  {
```

```

    { new Thread(new Runnable()
      { public void run()
        { for (int i=0;i<10;++i)
          System.out.println("Hello");
        }
      }).start();
    }
}

```

L'appel du constructeur de `MyThread()` est remplacé par une déclaration spéciale de la classe. Ce code est converti par le compilateur comme ceci :

```

class MaClass
{ void f()
  { class $1 implements Runnable
    { public void run()
      { for (int i=0;i<10;++i)
        System.out.println("Hello");
      }
    };
    new $1().start();
  }
}

```

Puis comme ceci :

```

class MaClass$1 implements Runnable
{
  public void run()
  { for (int i=0;i<10;++i)
    System.out.println("Hello");
  }
}
class MaClass
{ void f()
  { new Thread(new MaClass$1()).start();
  }
}

```

Une classe anonyme peut avoir des initialisations d'attributs mais ne peut pas avoir de constructeur.

```

new Runnable()
{ String msg="hello";
  String upper=msg.toUpperCase();
  ...
}.start();

```

Si la classe anonyme utilise des variables locales de la fonction `f()`, un constructeur est ajouté pour initialiser les variables privées de l'instance.

Il faut déclarer les variables de la méthode, utilisés par la classe anonyme, en `final`. Dans l'exemple suivant, la variable privée se nomme `max`.

```

class MaClass
{ void f(final int max)
  { new Thread(new Runnable()
    { public void run()
      { for (int i=0;i<max;++i)
        System.out.println("Hello");
      }
    }).start();
  }
}

```

Contrairement aux exceptions, cela ne fait pas partie de la signature de la méthode. Une sous-classe n'est pas obligée d'avoir le paramètre `max` en `final`.

```

class MaClassEx extends MaClass
{ void f(int max)
  {
  }
}

```

La méthode `MaClassEx.f(int max)` surcharge la méthode `MaClass.f(final int max)`. L'inverse est également vrai.

Le code de `MaClass` devient :

```

class MaClass$1 implements Runnable
{

```

```

    final int val$max;
    MaClass$1(int val$max)
    { this.val$max=val$max;
    }
    public void run()
    { for (int i=0;i<val$max;++i)
      System.out.println("Hello");
    }
  }
}
class MaClass
{ void f(final int max)
  { new Thread(new MaClass$1(max)).start();
  }
}

```

Il faut déclarer le paramètre `max` comme `final` car il est utilisé dans une méthode de la classe anonyme. L'instance anonyme peut exister après l'exécution de la méthode `f()`. Le traitement est identique avec les classes internes non anonymes. Toutes les variables locales à la méthode sont copiées par le constructeur de l'instance anonyme.

Une classe anonyme peut être déclarée en dehors d'une méthode. Pour cela, il faut utiliser la syntaxe permettant d'initialiser les attributs.

```

class MaClass
{ Object o=new Object()
  { bool f()
    { return o==null;
    }
  };
}

```

La visibilité dépend du contexte de création. Si la classe anonyme est déclarée lors de l'initialisation d'un attribut, elle peut consulter tous les attributs de l'instance externe.

Si la classe anonyme est déclarée lors de l'initialisation d'un attribut statique, elle ne peut consulter que les attributs statiques.

```

class MaClass
{ static Object o=new Object()
  { bool f()
    { return o==null;
    }
  };
}

```

La classe anonyme est dans ce cas une inner classe anonyme statique.

```

class MaClass
{ static class $1
  { bool f()
    { return o==null;
    }
  };
  static Object o=new $1();
}

```

Le code final ressemble alors à ceci :

```

static class MaClass$1
{ bool f()
  { return o==null;
  }
};
class MaClass
{
  static Object o=new MaClass$1();
}

```