

# La construction en Java

Philippe PRADOS

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)



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

## TABLE DES MATIERES

|    |   |   |
|----|---|---|
| 1. | Comment est construite une instance ? ..... | 3 |
| 2. | L'ordre de construction.....                | 3 |
| 3. | Conclusion .....                            | 6 |

## *Avant propos*

*Comment est réalisée la construction d'une instance avec Java ? Il est nécessaire de maîtriser des détails subtils pour éviter des erreurs dans les programmes.*

### 1. COMMENT EST CONSTRuite UNE INSTANCE ?

La construction d'une instance Java s'effectue en plusieurs étapes qu'il est intéressant de connaître. Cela peut avoir des effets de bords non maîtrisés. Je vous propose un petit exercice pour que vous puissiez vérifier si vous connaissez la construction en Java. Il faut deviner, sans l'exécuter, le mot de passe affiché par ce programme.

```
class Str
{ Str(char x)
  { System.out.print(x); }
};

class Base
{ static Str s=new Str('P');
  int a_=methode();
  final int b_;
  static
  { b_=surcharge();
  }
  Base()
  { System.out.print ('E');
  }
  int methode()
  { System.out.print ('L');
    return 0;
  }
  int surcharge()
  { System.out.print ('S');
    return 1;
  }
  static
  { System.out.print ('A');
  }
  static Str t;
  static
  { t=new Str('R');
  }
}

class Derive extends Base
{ int c_=surcharge();
  Derive()
  { super();
    System.out.print ('E');
  }
  int surcharge()
  { System.out.print ('L');
    return 2;
  }
  static
  { System.out.print ('A');
  }
}

public class Start
{ public static void main(String[] argv)
  { new Derive();
    System.out.flush();
  }
}
```

### 2. L'ORDRE DE CONSTRUCTION

La séquence des appels est indiquée ci-dessous dans les chiffres encadrés par des crochets.

```
class Base
{ static Str s=new Str('P'); // [1]
  int a_=methode();
  int b_=surcharge();
  Base()
  { System.out.print ('E'); // [7]
  }
  int methode()
```

```

    { System.out.print ('L'); // [5]
      return 0;
    }
    int surcharge()
    { System.out.print ('S');
      return 1;
    }
    static
    { System.out.print ('A'); // [2]
    }
    static Str t;
    static { t=new Str('R'); // [3]
    }
}

class Derive extends Base
{ int c_=surcharge();
  Derive()
  { super();
    System.out.print ('E'); // [9]
  }
  int surcharge()
  { System.out.print ('L'); // [6] et [8]
    return 2;
  }
  static
  { System.out.print ('A'); // [4]
  }
}

```

Le mot de passe est « PARALLELE ». Cela permet de comprendre comment java charge une classe et construit une instance. La construction est décomposée en plusieurs étapes.

1. Chargement de la classe et initialisation des attributs statiques
  - a. Initialisation de la super-classe.
  - b. Initialisation des attributs statiques et appel de la méthode d'initialisation de classe dans l'ordre de déclaration.
2. Allocation de la mémoire nécessaire à l'instance
3. Initialisation de tous les attributs avec leurs valeurs par défaut et appel des blocs anonymes pour initialiser les attributs `final`.
4. Validation du polymorphisme
5. Appel des constructeurs
  - a. Appel de `super()`
  - b. Appel des initialisations par défaut des attributs dans l'ordre de déclaration
  - c. Appel du corps du constructeur

Cela a plusieurs conséquence :

1. L'ordre de déclaration de la méthode `static {}` vis à vis des attributs statiques est important.
2. Si un constructeur appelle une méthode qui est surchargée, c'est la version surchargée qui est exécutée (contrairement au C++). Attention, l'instance dérivée n'est pas encore construite ! La méthode surchargée ne peut utiliser que les attributs hérités. Il ne faudrait pas appeler de méthode non `final` dans un constructeur.
3. L'initialisation des attributs s'effectue juste après l'appel du `super()` et avant le corps du constructeur.
4. L'ordre de déclaration des attributs est important.

Java permet d'avoir plusieurs blocs statiques.

```

class Base
{
  static
  { ...
  }
  static
  { ...
  }
}

```

Initialiser une variable statique est équivalent à rédiger plusieurs blocs statiques.

```

class Base
{ static Str s=new Str('P');
  static
  { System.out.print ('A');
  }
}

```

```

}
}

```

est équivalent à :

```

class Base
{ static Str s;
  static
  { s=new Str('P');
  }
  static
  { System.out.print ('A');
  }
}

```

Le compilateur réunit tous les blocs statiques pour en faire une méthode statique appelée `<clinit>`.

```

class Base
{ static Str s;
  static void <clinit>()
  { // Premier bloc
    { s=new Str('P');
    }
    // Deuxieme bloc
    { System.out.print ('A');
    }
  }
}

```

Il est possible d'initialiser des attributs `final` dans un bloc statique. Cela permet d'effectuer des calculs complexe lors de l'initialisation de l'attribut. Si l'attribut final est `static`, le bloc qui l'initialise sera ajouté à l'initialisation de la classe. Si l'attribut final n'est pas `static`, il sera ajouté à tous les constructeurs. Cela peut avoir des conséquences subtiles.

Quel est la différence entre les deux versions suivantes ?

```

class Base
{ static final int i=10;
}

class Base
{ static final int i;
  static
  { i=10;
  }
}

```

Les deux versions initialise l'attribut final `i` avec la valeur dix. Mais, dans le premier cas, si une classe utilise la constante `Base.i`, le compilateur utilise directement la valeur dix à la place.

```

class AutreClasse
{ void m()
  { int j=Base.i;
  }
}

```

Ce code est compilé comme ceci :

```

class AutreClasse
{ void m()
  { int j=10;
  }
}

```

Cela améliore les performances car il n'est plus nécessaire de consulter la variable `Base.i`. Par contre, si la valeur de la constante évolue, et que l'on ne recompilait pas la classe `AutreClasse`, le code sera erroné.

```

class Base
{ static final int i=15;
}
class AutreClasse
{ void m()
  { int j=10; // !!!
  }
}

```

Ce problème n'existe que pour les attributs `static final` initialisés et de type primitif. La deuxième version la classe `Base` permet d'éviter cet inconvénient. Toutes les classes utilisant la variable `Base.i` consulteront sa valeur. Lors de la rédaction d'un framework, il est important de gérer cela. Les attributs `final` devraient être initialisés dans un bloc séparé.

Java ne permet pas d'appeler directement la version d'une méthode déclarée dans la classe courante. Il propose `super` pour appeler la version héritée ou un appel direct pour appeler la version surchargée. Si la classe est entre deux classes, elle ne peut pas appeler sa propre version.

```
class A
{
    void methode()
    { }
}
class B extends A
{
    B()
    {
        methode(); // ???
    }
    void methode()
    {
    }
}
class C extends B
{
    void methode()
    {
    }
}
```

Comment le constructeur de `B()` peut-il appeler la version `B.methode()` ? S'il utilise `methode()`, il appelle la version de `C.methode()`. S'il utilise `super.methode()`, il appelle la version `A.methode()`. Il n'est pas possible d'indiquer avec précision, la version à utiliser.

Au niveau de la machine virtuelle de java, l'appel d'une méthode à l'aide `super.methode()` entraîne l'utilisation de `invokespecial A.methode`. L'invocation sans le préfixe `super` entraîne l'utilisation de `invoke C.methode`. Il faudrait un appel à `invokespecial B.methode`. Pour cela, il faut modifier le langage java pour lui permettre d'interpréter un code du type `B.methode()` comme un appel via `invokespecial` si la méthode n'est pas statique.

Pour appeler la version locale d'une méthode, il faut la décomposer en deux parties.

```
class Base
{
    ...
    int b_=surcharge_();
    ...
    private final int surcharge_()
    { System.out.print("mange un ");
      return 1;
    }
    public int surcharge()
    { return surcharge_();
    }
}
```

- Une version `private final` ne peut pas être surchargé,
- et une version `public`, surchargeable, appelle la version `private`.

Un attribut ou le constructeur peut appeler la version `private` de la méthode. La vitesse du code n'est pas impactée par cette écriture car le corps de la méthode `surcharge_()` est recopié par le compilateur dans la méthode `surcharge()`.

### 3. CONCLUSION

Pour résumer, on peut définir quelques règles pour rédiger un constructeur.

- L'ordre de rédaction des initialisations à un impacte sur l'ordre d'exécution.
- Il ne faut pas appeler de méthode non `final` dans un constructeur.
- Il ne faut pas initialiser d'attribut `static` dans un constructeur.