

La méthode toString()



Ce document explique l'utilisation et la rédaction de la méthode toString(). Il explique également comment fonctionne la concaténation de chaîne de caractère.

Par [Philippe PRADOS](http://www.prados.fr) - 1999
www.prados.fr

La méthode toString() permet d'afficher une instance. Elle est proposée dans la classe Object pour faciliter le débogage. On croit souvent, à tort, que cette méthode est appelée implicitement lorsque cela est nécessaire. C'est une erreur. En effet, le compilateur Java ne connaît pas l'existence de cette méthode. On peut la supprimer de la classe Object. En fait, le compilateur ne connaît qu'une chose : comment traduire l'opérateur plus lorsqu'une des instances est du type String.

Addition de chaînes

Lorsqu'on additionne une String avec une autre instance, le compilateur génère la création d'un objet StringBuffer.

```
String s="world";  
"Hello "+s;
```

La dernière ligne est traduite automatiquement par le compilateur de Sun ou d'IBM en :

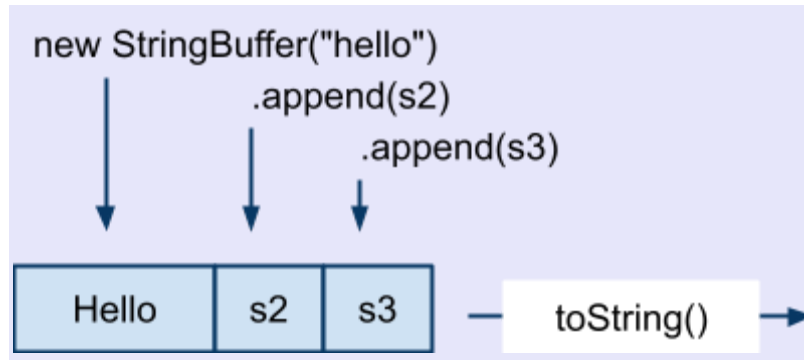
```
new StringBuffer("Hello ").append(s).toString();
```

Le compilateur construit une instance StringBuffer avec le premier membre de l'addition, puis, appelle la méthode append() pour tous les éléments suivants, et enfin, convertit le StringBuffer en String. La seule chose qu'il connaît est l'existence de la méthode toString() de la classe StringBuffer.

Il est à noter que si on additionne deux chaînes de caractères constantes, le compilateur optimise le code. "Hello "+"world" devient "Hello world". Vous pouvez utiliser cela pour formater votre source, et découper une chaîne longue en plusieurs segments.

Si on additionne plusieurs chaînes, les méthodes append() sont enchaînées. "Hello "+s2+s3 devient

```
new StringBuffer("Hello ").append(s2).append(s3).toString();
```



Le compilateur de Microsoft propose une traduction légèrement différente. Il construit une instance `StringBuffer()` vide, et ajoute tous les membres de l'opération.

```
new StringBuffer().append("Hello ").append(s2).append(s3).toString()
```

Pour additionner une chaîne avec un type primitif, la classe `StringBuffer` propose différentes versions de la méthode `append()`.

```
char c='a';
"Hello "+c
```

devient

```
new StringBuffer("Hello ").append(c).toString()
```

Il existe une version de la méthode `append()` qui reçoit un `Object`. C'est cette méthode qui va appeler la méthode `toString()` de son paramètre.

```
Date d=new Date();
"Hello "+d
```

devient :

```
new StringBuffer("Hello ").append(d).toString();
```

La méthode `append(Object)` appelle la méthode `toString()` de `d` et ajoute la chaîne au tampon. C'est ce mécanisme qui fait croire que le compilateur appelle automatiquement la méthode `toString()`.

Si le membre à gauche d'une addition n'est pas une constante `String`, le compilateur de Sun converti l'instance à gauche à l'aide des méthodes `String.valueOf()`.

'a'+s devient

```
new StringBuffer(String.valueOf('a')).append(s).toString()
```

Il existe une version de la méthode `valueOf()` qui appelle la méthode `toString()` d'un `Object`.

Le modèle utilisé par Microsoft évite l'invocation de la méthode `valueOf()`.

```
new StringBuffer().append('a').append(s).toString();
```

Le compilateur IBM utilise une approche intermédiaire. Si le paramètre à gauche est du type String, le compilateur génère l'appel à la méthode `valueOf()` pour gérer le cas où le paramètre est à null.

```
new StringBuffer(String.valueOf(s1)).append(s2).toString();
```

Si le paramètre à gauche est d'un type différent, il utilise l'approche de Microsoft.

Ces différents choix de traduction ont un impact sur la rédaction d'une expression. En effet, est-il préférable d'encadrer un caractère d'apostrophe ou de guillemet ? Est-il préférable d'écrire `'a'+s` ou `"a"+s` ? Regardons comment Sun, IBM et Microsoft traduisent ces expressions.

Pour Sun, cela donne :

```
new StringBuffer(String.valueOf('a')).append(s).toString();
new StringBuffer("a").append(s).toString();
```

Pour IBM, cela donne

```
new StringBuffer().append('a').append(s).toString();
new StringBuffer("a").append(s).toString();
```

Pour Microsoft, cela donne

```
new StringBuffer().append('a').append(s).toString();
new StringBuffer().append("a").append(s).toString();
```

Pour le compilateur de Sun, il est préférable d'utiliser une chaîne de caractère pour le premier paramètre, car sinon, la méthode `String.valueOf()` en construira une dynamiquement. Cela n'est pas le cas pour les caractères placés à droite.

`s+'a'` devient

```
new StringBuffer(String.valueOf(s)).append('a').toString();
```

Avec les approches d'IBM et de Microsoft, il est toujours préférable d'utiliser le caractère à la place de la chaîne.

Par principe, il ne faut pas additionner de chaîne composée d'un seul caractère. Il est plus rapide d'ajouter un seul caractère à la suite du tampon de `StringBuffer` que de faire une boucle pour ajouter tous les caractères de la chaîne (qui n'en contient qu'un).

Par contre, ajouter une constante numérique à une chaîne n'est pas efficace.

```
s + 123
```

Il est préférable d'ajouter directement une chaîne de caractère.

```
s + "123"
```

Cela évite au programme de convertir la constante en chaîne. Cet algorithme

est en effet coûteux.

Si on désire construire une chaîne avec des types primitifs, il faut utiliser explicitement la méthode `valueOf()` afin d'avoir une chaîne dans l'expression.

```
int  
i=2;  
char c='a';  
String s=String.valueOf(i)+c
```

La classe `StringBuffer` est capable d'augmenter la taille de son tampon lorsque cela est nécessaire. Par défaut, la taille du tampon est de 16 caractères. Lors de l'ajout du dix-septième, un nouveau tampon de taille double est créé, le premier est copié dans le second avant d'être abandonné. Il est judicieux d'estimer la taille que prendra la chaîne complète afin d'initialiser la taille du tampon en conséquence.

Pour synthétiser :

- Si la constante à ajouter est du type caractère, utiliser les apostrophes (`s + 'a'`)
- Si la constante à ajouter n'est pas du type caractères, utiliser les guillemets (`s + "123"`)

StringBuffer.toString()

La méthode `toString()` de la classe `StringBuffer` doit théoriquement dupliquer son tampon dans une nouvelle instance immuable de type `String`. En effet, le tampon nécessaire à une instance `String` ne doit pas être modifiable.

```
public class StringBuffer  
{  
    private char[] buf_;  
    ...  
    public String toString()  
    {  
        return new String(buf_);  
    }  
}
```

Pour éviter la duplication du tampon, alors que la durée de vie d'une instance `StringBuffer` est généralement courte, la méthode `toString()` est rédigée différemment.

```
public class StringBuffer  
{  
    private char[] buf_;  
    ...  
    public String toString()  
    {  
        return new String(this);  
    }  
}
```

L'instance String est construite à partir de l'instance StringBuffer courante. En fait, la classe String récupère directement le tampon et signale à l'instance StringBuffer qu'il est dorénavant partagé. La classe String invoque la méthode StringBuffer.setShared() qui n'est pas publique. Ce comportement n'est possible que si les classes String et StringBuffer sont dans le même package.

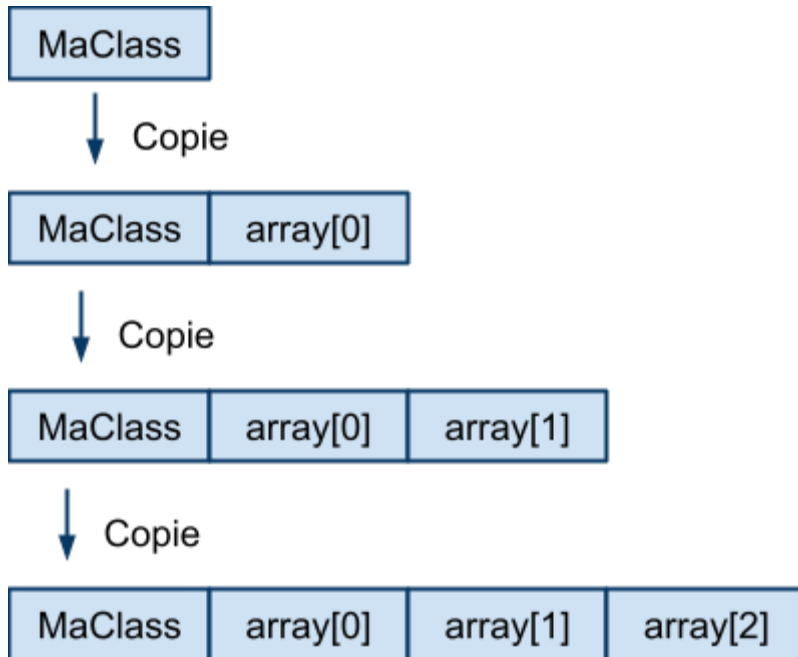
Si une méthode de l'instance StringBuffer doit modifier le tampon et que celui-ci est partagé, une copie est effectuée avant la modification. Cela permet de supprimer le partage et garantit que les instances String sont immuables.

Opérateur +=

Pour construire une chaîne de caractères, on peut utiliser l'addition proposée par Java, mais il est parfois préférable de manipuler directement une instance StringBuffer. Par exemple, si la chaîne doit être construite dans une boucle.

```
public class MaClass
{
    private int att_;
    private Object[] array_;
    public String toString()
    {
        String r="MaClass
        {
            att_="+att_+" array_=";
            for (int i=0;i<array_.length;++i)
            {
                r+=array_[i];
            }
            return r;
        }
    }
}
```

L'appel de `r+=array_[i]` entraîne la création d'un StringBuffer et la duplication du tampon à chaque cycle.



```
r=new StringBuffer(r).append(array_[i]).toString();
```

L'expression `x += y` est traduite par le compilateur en `x = x + y`. Cet opérateur est un mauvais exemple d'utilisation des chaînes de caractères. Il faut convertir `+=` en utilisant un `StringBuffer`. Malheureusement, il n'est pas possible de récupérer l'instance temporaire construite par le compilateur.

```
StringBuffer b="MaClass { att_="+att_+" array_="; // Bug
```

Le code généré est le suivant

```
StringBuffer b=new StringBuffer("MaClass { att_=")
    .append(att_)
    .append(" array_=")
    .toString();
```

Il suffirait de supprimer l'appel à la méthode `toString()`. Pour le moment, ce n'est pas dans la définition du langage. Il faut alors, soit convertir la chaîne en `StringBuffer` avant de traduire l'opérateur `+=`,

```
String s="MaClass { att_="+att_+" array_=";
StringBuffer r=new StringBuffer(s);
```

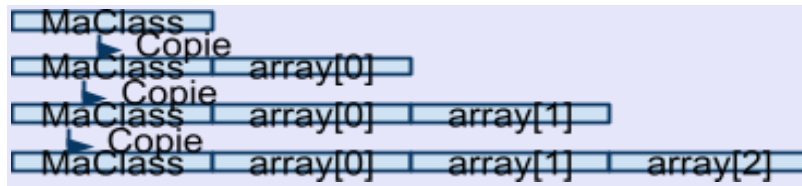
soit traduire directement à la main l'addition.

```
StringBuffer r=new StringBuffer("MaClass { att_=")
    .append(att_)
    .append(" array_=");
```

Cette deuxième approche est plus complexe à rédiger, mais plus efficace en terme de mémoire et de vitesse. Ensuite, il faut ajouter les différentes chaînes dans la boucle.

```
for (int i=0;i<array_.length;++i)
```

```
{
    r.append(array_[i]);
}
```



Il ne faut pas oublier de convertir le StringBuffer en String avant le retour de la méthode. Le code complet devient :

```
public class MaClass
{
    private int att_;
    private Object[] array_;
    public String toString()
    {
        StringBuffer r=new StringBuffer("MaClass { att_=")
            .append(att_)
            .append(" array_=");
        for (int i=0;i<array_.length;++i)
        {
            r.append(array_[i]);
        }
        return r.toString();
    }
}
```

Ce code est un peu complexe à lire, mais offre le maximum de performance. Il ne faut pas utiliser l'opérateur += avec une String.

Conversion implicite

Java ne sait pas convertir implicitement un Object en String.

```
public class MaClass
{
    private String nom_;
    public String getNom()
    {
        return nom_;
    }
    public void setNom(String x)
    {
        ...
    }
}
...
Object s=new String("abc");
MaClass obj=new MaClass();
obj.setNom(obj); // Ne compile pas
```

Si vous le voulez, vous pouvez offrir cette conversion. Au lieu de proposer la

méthode `setNom()` recevant une `String` en paramètre, indiquez la classe `Object`.

```
public class MaClass
{
    ...
    public void setNom(Object x)
    {
        nom_=x.toString();
    }
}
```

C'est le modèle qu'utilisent les méthodes `String.valueOf()` et `StringBuffer.append()`. La méthode `valueOf(Object)` retourne la chaîne "null" si le paramètre est à null et appelle la méthode `toString()` sinon. Cela impose à la méthode `toString()` de ne jamais retourner null.

Comment rédiger la méthode toString()

La méthode `toString()` à un statut particulier. Elle peut servir à afficher une information à l'utilisateur à partir d'une instance, ou bien elle sert à déverminer le programme. Par exemple, le retour de la méthode `String.toString()` est généralement présenté à l'utilisateur. Par contre, des objets métiers vont afficher une trace de l'instance.

Sémantiquement, la méthode `toString()` permet de convertir une instance en chaîne de caractère. Une adresse peut par exemple être convertie en chaîne de caractère pour indiquer les coordonnées postales tel qu'imprimé sur l'enveloppe. La méthode `toString()` n'est pas NLS (National Language Support). Elle retourne une valeur codée généralement en dure. La construction d'une adresse est différente suivant les pays. En France par exemple, l'adresse est construite dans un ordre inverse de ce qui est nécessaire à l'acheminement du courrier. On commence par le nom du destinataire et l'on finit par sa ville. L'information principale pour le tri est le code postal, confirmé par la ville, et en dernier, le nom de la personne. D'autres pays, on choisit un formatage différent.

Convertir une instance en `String` n'est pas possible si on désire respecter la philosophie de java en étant international. On peut imaginer que la méthode `toString()` retourne une chaîne correspondant à la localisation courante. A défaut, il faut généralement rédiger cette méthode dans une optique de déverminage. D'autres méthodes NLS serviront à présenter un objet à l'utilisateur. Par exemple, il ne faut pas afficher le retour de la méthode `toString()` de la classe `Date`. Il faut utiliser la classe `java.text.DateFormat` à la place.

Pour afficher les attributs primitifs, il faut utiliser l'addition de chaîne de caractères.

```

public class MaClass
{
    private int age_;
    public String toString()
    {
        return "MaClass{ age_="+age_+"}";
    }
}

```

Pour afficher les types complexes, il faut bien identifier les différentes catégories de références. Une référence peut représenter une agrégation ou une relation. Il faut demander l'affichage complet de l'instance agrégée, mais un résumé de l'instance en relation. Sinon, on s'expose à des récursivités infinies.

```

public class Employé
{
    private Adresse agr_;
    private Entreprise rel_;
    public String toString()
    {
        return "Employé{ agr_="+agr_+
            ", rel_="+rel_.getNom()+
            "'}";
    }
}

```

De plus, il faut garantir que la méthode `toString()` ne retournera jamais la valeur `null`.

L'implémentation par défaut de la méthode `toString()` retourne une chaîne de caractère construite à partir du nom de la classe et de la valeur de hash de l'instance.

```

public class Object
{
    ...
    public String toString()
    {
        return getClass().getName() +
            '@' + Integer.toHexString(hashCode());
    }
}

```

Cela permet d'identifier l'instance. Vous pouvez utiliser une approche similaire pour afficher les relations.

```

public class Employé
{
    ...
    public String toString()
    {
        return "Employé{ agr_="+agr_+
            ", rel_="+rel_.getClass()+
            "'@'+
            Integer.toHexString(rel_.hashCode())

```

```

        '}' ;
    }
}

```

Vous pouvez également utiliser la méthode `System.identityHashCode(Object)`. Cela permet d'avoir une valeur déduite de l'adresse mémoire de l'instance.

```

public class Employé
{
    ...
    public String toString()
    {
        return "Employé{ agr_="+agr_+
            ", rel_="+rel_.getClass()+
            '@'+Integer.toHexString(
                System.identityHashCode(rel_))
            '}' ;
    }
}

```

Lors d'un héritage, il peut être nécessaire d'appeler la méthode `toString()` héritée.

```

public class Humain
{
    ...
    public String toString()
    {
        return "Humain{ nom_="+nom_+'}' ;
    }
}

public class Employé extends Humain
{
    ...
    public String toString()
    {
        return "Employé{ "+super.toString()+
            " employeur_="+employeur_.getNom()+
            '}' ;
    }
}

```

En résumé :

- La méthode `toString()` doit être rédigée dans une optique de déverminage ;
- Elle ne doit pas retourner `null` ;
- Il ne faut pas utiliser l'opérateur `+=` pour construire une chaîne. Utilisez plutôt un `StringBuffer` ;
- L'affichage des relations doit être différent de l'affichage des agrégations ;
- Encadrez les constants caractères d'apostrophes et non d'accolades.