

Le type de null

Philippe PRADOS

pp@philippe.prados.name

Avant-propos

Avec java, quel peut bien être le type de `null` ? Le compilateur propose des optimisations qui apparaissent parfois dans le code.

La valeur `null` peut être affectée à toutes les références. Quel est son type ? Si `null` est du type `Object`, cela oblige à convertir toutes les valeurs `null` avant une affectation.

```
String s=(String)null;
```

Sans la conversion, le compilateur refuserait la compilation. Pour éviter cela, les concepteurs de Java n'ont pas donné de type à `null`. Le compilateur converti implicitement la valeur `null` dans le type du pointeur participant à l'expression. Cela fonctionne dans la majorité des situations.

Toutefois, dans certains cas, cela ne fonctionne pas. S'il existe plusieurs méthodes de mêmes noms attendant une référence de type différent, le compilateur est capable de choisir la bonne méthode.

```
class A
{ public void f(String x)
  { ...
  }
  public void f(Object x)
  { ...
  }
};
```

Suivant le type de pointeur du paramètre, la méthode `f(String)` ou `f(Object)` sera appelé. Attention, cette sélection sera faite par rapport au type du pointeur, mais pas par rapport à l'instance réellement pointée.

```
{ String s="hello";
  Object o=s;
  A a=new A();
  a.f(s); // Appel f(String)
  a.f(o); // Appel f(Object)
}
```

Même si les deux pointeurs `s` et `o` pointent sur la même instance, l'appel sera différent.

Si maintenant on appelle la méthode `f()` avec la constante `null` ? Quelle version sera appelée ?

```
a.f(null);
```

Le compilateur cherche à convertir la valeur `null` en remontant l'arbre d'héritage de plus en plus. Parmi les méthodes candidates, il va alors trouver en premier la méthode `f(String)`. Il appellera donc cette version.

Mais s'il existe deux méthodes `f()` avec des types de paramètres n'offrant pas d'héritage entre eux ?

```
class A
{ public void f(String x)
  { ...
  }
  public void f(Date x)
  { ...
  }
};
```

Dans ce cas, le compilateur n'est pas capable de lever l'ambiguïté. Il refusera la compilation.

```
a.f(null); // Ambiguity
```

Il est possible de lever l'ambiguïté en castant la constante `null`.

```
a.f((String)null);  
a.f((Date)null);
```

Il est déconseillé d'avoir deux méthodes de mêmes noms recevant chacun le même nombre d'objet. Sinon, le client de la classe doit être rigoureux lors de l'appel de la méthode avec la constante `null`.

Cela est particulièrement vrai si les paramètres sont de types héritant entre eux. Il faut vérifier les paramètres pour garantir une exécution correcte.

```
class A  
{ public void f(String x)  
  { ...  
  }  
  public void f(Object x)  
  { if (x instanceof String) f((String)x);  
    ...  
  }  
};
```

Offrir ces deux méthodes permettent une optimisation du code dans le cas où le paramètre est du type `String`. Si le code ne propose pas d'optimisation, seule la version `f(Object)` doit être proposée.

Ce problème n'existe pas avec les types primitifs.

```
class A  
{ public void f(String x)  
  { ...  
  }  
  public void f(int x)  
  { ...  
  }  
  public void f(float x)  
  { ...  
  }  
};
```

Il n'y a pas d'ambiguïté possible avec les types primitifs car les constantes sont typées.

```
a.f(1);  
a.f(1.0);
```