Java

CM3-3 : Java, héritage

Mickaël Martin Nevot

V1.1.0

Java

Variable/méthode d'instance

- **Varie d'une instance (objet) à l'autre**
- Initialisation non obligatoire. Valeur par défaut :
  - Entier `0`
  - Flottant `0.0`
  - Booléen, caractère `false`
  - Référence `null`
- Méthode d'instance (passage par valeur) :
  - Type primitif : ne modifie pas la valeur d'une variable
  - Référence : c'est l'objet qui est modifié par la référence

Notation pointée et this

- Notation pointée :
  - Pour spécifier une hiérarchie de paquetages :

```
package package1.package2;
import java.lang.*;
```

  - Pour accéder à un attribut :

```
myObj.att1; // Accès à un attribut.
```

  - Pour accéder à une méthode :

```
myObj.meth1(); // Accès à une méthode.
myObj.meth2(1, 2); // Accès à une méthode.
```

- Mot clef `this` (lecture seule) :
  - Désigne l'**objet courant** (celui dans lequel on code) :

```
this.att1; // Accès à un attribut.
this.meth1(); // Accès à une méthode.
```

  - Fortement conseillé même si le sens n'est pas équivoque

Paquetage

- Groupe de classes associé à une fonctionnalité
- À chaque paquetage correspond un répertoire
- Un paquetage peut contenir un autre paquetage
- Mot clef `package` (première instruction d'une classe) :

```
package package1.package2;
// MyClass appartient au paquetage package2
// qui appartient lui-même à package1.
class MyClass { … }
```

- Mot clef `import` :

```
// Importe la classe MyClass (ci-dessus).
import package1.package2.MyClass;
// Importe tous les éléments du paquetage lang
// qui appartient lui-même au paquetage java.
import java.lang.*;
```

Affectation, recopie, comparaison

- Affectation/recopie : =
  - Type primitif : modification distincte
  - En cas de recopie, les deux opérandes restent distincts
  - Objet : modification commune :

```
a = 8; // Affectation.
b = c; // Recopie (type primitif ou référence).
```

- Recopie de contenus d'objets : `clone()`
  - Modification distincte :

```
a = b.clone();
```

- Comparaison : ==, `equals(Objet o)`

```
a == 8; // Comparaison.
b == c; // Comparaison (type primitif ou référence d'objet).
a.equals(b) // Comparaison de contenus d'objets.
```

Surcharge

- Possible pour n'importe quelle méthode
- Méthode avec le même nom mais pas avec la même liste de paramètres (**signature non identique**)
- Type de retour possiblement différent si les listes des paramètres sont différentes

```
int   meth1()         { … }
int   meth1(int p)    { … }
float meth1(float p)  { … }
float meth1(int p)    { … } // Impossible !
```

Le type de retour ne fait pas partie de la signature

Constructeur

- **Même nom que la classe**
- Méthode spéciale appelée à l'**instanciation d'un objet** pour initialiser son état
- **Pas de valeur de retour**
- Si aucun constructeur n'existe, un **constructeur par défaut** (sans paramètre et qui ne fait rien) est défini
- Mot clef `new` (création et allocation mémoire)

```
class MyClass { // Déclaration de classe.
    MyClass(int p) { … }
    …
}
…
MyClass myObj1; // Déclaration.
// Création et allocation mémoire : instanciation.
myObj1 = new MyClass(5);
```

Constructeurs multiples

- Surcharge de constructeurs
- Mot clef `this`

```
class MyClass {
    MyClass() {
        this(5, new Objet( … )); // Qu'en première instruction d'un constructeur.
    }
    MyClass(int p) {
        this(p, new Objet( … ));
    }
    MyClass(int p, Objet a) {
        instruction1
         …
    }
    …
}
```

Encapsulation

- Concerne : classe, constructeur, attribut et méthode
- Accessibilité :
  - `public` : accessible de partout et sans aucune restriction (la classe principale doit nécessairement être publique)
  - `protected` : accessible aux classes du paquetage et à ses classes filles
  - `private` : accessible uniquement au sein de sa classe
  - (par défaut) : accessible aux classes du paquetage
- Accesseurs/mutateurs :

```
private int cpt;
public int getCpt(){ return this.cpt; }
public void setCpt (int p){ this.cpt = p; }
```

Héritage

- Héritage simple (une seule super-classe et unidirectionnelle)
- Mot clef `extends`
- Surcharge
- Redéfinition :

Ne pas confondre !

  - Même signature de méthode (ne peut pas être moins accessible)
  - Réécriture du code
  - Redéfinition d'attributs (les attributs redéfinis sont ajoutés)

```
class MySuperClass  {
    int meth1(int a) { instruction1 }
}
…
class MyClass extends MySuperClass {
    int meth1(int a) { instruction2 }
}
```

super : méthode

- Permet de **réutiliser le code de la méthode de la super-classe** :

```
class MySuperClass  {
    meth1(int a) { instruction1 }
}
…
class MyClass extends MySuperClass {
    meth1(int a) {
        …
        super.meth1(a);
        …
    }
}
```

Constructeur : super()

- Permet de **réutiliser le code du constructeur de la super-classe** :
  - Doit être la **première instruction**
  - Appel implicite `super()` (sans paramètre) par défaut
  - Pas compatible avec `this()`

```
class MySuperClass  {
    MySuperClass (int a) { instruction1 }
}
…
class MyClass extends MySuperClass {
    MyClass(int a) {
        super(a - 1);
        …
    }
}
```

S'il y a au moins un constructeur explicite avec au moins un paramètre dans la super-classe : il y a désactivation du constructeur par défaut et un appel implicite de ce dernier dans la classe-fille générera une erreur

Statique

- **Variable de classe** :
  - Donnée commune à tous les objets d'une même classe
  - Existe même s'il n'y a aucune instance de la classe :

```
public static int att1; // Définition.
…
MyClass.att1 = 3;
myObj.att1 = 3; // Fortement déconseillé !
```

  - Constante de classe :

```
public static final int att1 = 3; // Définition et initialisation.
```

- **Méthode de classe** :
  - Ne s'intéresse pas à un objet particulier :

```
public static int meth1() { … }; // Définition.
…
MyClass.meth1();
myObj.meth1(); // Fortement déconseillé !
```

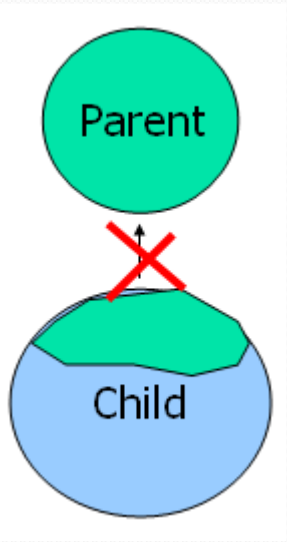
Mot clef final

- Constante
- Constante de classe
- Méthode (pour interdire toute redéfinition) :

```
public final int meth1() { … }
```

- Classe (pour interdire tout héritage) :

```
public final class MyClass { … }
```



Types scellés JAVA 17+

- Limite l'héritage : moins restrictif que `final`
- Indique quels sont les types de données pouvant hériter
- Mots clefs : `sealed, non-sealed, permits`

```
public sealed class Shape permits Circle, Square { … }
…
public final class Triangle extends Shape { … } // Interdit.
…
public non-sealed class Circle extends Shape { … } // Autorisé.
…
public final class ColoredCircle extends Circle { … } // Autorisé.
…
public sealed interface FormFactor permits Triangle, Rectangle { // Autorisé.
```

Crédits

Auteur

Mickaël Martin Nevot
mmartin.nevot@gmail.com

Carte de visite électronique

Relecteurs

Cours en ligne sur : www.mickael-martin-nevot.com