

JAVA SE - Portée package et protected

Les modificateurs de "portée package" et protected

Les mécanismes d'encapsulation permettent de mettre en place de véritables "cercles de responsabilité". Un programmeur est responsable du code d'une classe et garde sous son contrôle exclusif tout ce qui est marqué `private`. Il offre à d'autres programmeurs des accès à tout ce qui est marqué `public`.

Dans le cadre d'une politique de composants logiciels ces programmeurs "client" ne sont pas forcément connus du programmeur responsable du code. On a donc au travers de l'API publique d'un code un engagement fort : voici des codes dont le fonctionnement est garanti (du moins on peut l'espérer) et dont la durée de vie est également garantie (on n'a pas le droit de rendre une A.P.I. "publiée" inopérante : tout juste peut-on rendre des éléments `@Deprecated`).

Pour la suite de cette note nous allons considérer que les méthodes publiques sont des "services" rendus par la classe, alors que des méthodes d'instance privées relèvent de "mécanismes internes" à la classe. La distinction est artificielle (il est impossible de tracer une frontière entre les deux concepts) mais va nous servir à mieux expliquer les principes des modificateurs de portée.

Eléments en portée dans le package

Entre le code privé et le code public, il existe un niveau de responsabilité intermédiaire qui est celui de la portée dans le *package*. Dans un package on trouve un ensemble de codes qui concourent à un but commun. Il est possible d'avoir des codes qui ont un certain niveau d'interdépendance entre eux.

On pourrait dire que si, à un instant donné, une classe est sous la responsabilité d'un programmeur, le package relève lui de la responsabilité d'une équipe (donc de programmeurs qui sont censés se connaître et travailler de concert).

Au premier niveau de déclaration d'une classe tous les éléments qui ne sont marqués ni `private`, ni `public`, ni `protected` sont en portée dans le *package* (on dit parfois "*friendly*").

Quelques exemples:

- Une méthode statique en portée dans le package exprime un "service" partagé par les codes locaux. On a mutualisé un code pour les besoins communs des codes du package.

On pourrait imaginer que plusieurs classes indépendantes d'un même package ont besoin d'effectuer le même calcul. Celui-ci s'effectue dans un code partagé mais qui n'est pas exposé à l'extérieur car il ne concerne que ces classes et doit pouvoir évoluer à discrétion.

C'est en effet une différence supplémentaire avec les services public : on a tout à fait le droit de modifier la signature de ce service ou de le supprimer. Les seuls programmes impactés sont locaux !

- Sur le même principe on peut avoir des classes en portée dans le package pour centraliser un ensemble de services "privés" à destination de l'équipe.

```
class ServicesCommuns {  
    static final String nomPackage =  
ServicesCommuns.class.getPackage().getName();  
    static final Logger LOGGER = Logger.getLogger(nomPackage) ;  
    // recherche d'autres ressources liées au package  
    // services communs, etc...  
}
```

(On n'a pas forcément que des services statiques : on peut avoir des classes dont les instances

ne sont créées que par les codes du package - avec l'avantage que des versions ultérieures du package peuvent s'accompagner de modifications substantielles, ou de disparition, de ces classes-).

- Il est possible de disposer de champs ou de méthodes d'instance *friendly* : par ce biais la classe expose aux autres codes du package des "mécanismes" internes. La classe `java.awt.Component` expose ainsi des champs `x`, `y` (coordonnées) `width`, `height` (dimensions) ce qui simplifie les opérations de disposition de ce `Component` pour les codes du package `java.awt`.

Eléments protected

Il est possible d'étendre l'accès de membres ou constructeurs d'une classe au delà du cercle des codes du package en utilisant le modificateur `protected`.

Il y a là une propriété souvent mal comprise : ce qui est `protected` est aussi *friendly*! C'est pourtant logique quand on considère la hiérarchie des cercles de responsabilité: `private` (programmeur), *friendly* (groupe), `protected` (sous-classes dans d'autres packages).

Les modalités de l'utilisation des éléments `protected` est toutefois particulière. Il est faux de dire que ce qui est `protected` est utilisable dans toute sous-classe.

Voici un contre-exemple :

```
package org.demos.generation_x;
```

```
public class Mere {  
    protected int valeur ;  
}
```

et une autre classe dérivée dans un autre *package* :

```
package org.demos.generation_y;
```

```
public class Fille extends org.demos.generation_x.Mere {  
    public void modifieValeur( Mere autre) {  
        autre.valeur = this.valeur ; // _compiler_error_  
        // idem pour: this.valeur = autre.valeur  
    }  
}
```

Ce code **ne se compile pas!**

En effet si `this.valeur` est "accessible" dans l'instance courante `autre.valeur` ne l'est pas. Si cela était possible alors la classe `org.demos.generation_z.Fils` qui hérite de `Mere` serait exposée à une consultation ou une modification par le code de `Fille` qui est dans un package complètement différent. (par `uneFille.modifieValeur(unFils)`).

Les règles d'utilisation de `protected` varient en fonction du fait que l'élément concerné est lié à une instance ou ne l'est pas (static) :

- Il est possible d'avoir une variable membre `protected` mais cela a l'inconvénient de limiter l'encapsulation. **Attention** : ce qui est `protected` fait partie de l'API et donc ne peut être modifié de manière substantielle dans une version ultérieure!

Pour un exemple voir le champ `modCount` de `AbstractList`

Un autre exemple :

```
public class VoitureCustomisee {  
    protected final Voiture base ;  
    public VoitureCustomisee(Voiture base,°°°) {  
        this.base = base ;  
        // etc...  
    }  
    public Voiture getBase() { return this.base ; }  
    // autres codes  
}
```

Et maintenant dans un autre package :

```
class VoitureAncienneCustomisee extends VoitureCustomisee {  
    public VoitureAncienneCustomisee(VoitureAncienne base, °°°°) {  
        // VoitureAncienne extends Voiture  
        super(base, °°°) ;  
        // etc...  
    }  
    public VoitureAncienne getBase() {  
        return (VoitureAncienne) this.base ;  
    }  
}
```

(On remarquera tout de même que l'exposition de la variable base n'est pas strictement nécessaire et qu'on aurait pu faire autrement)

Comme pour tout ce qui est protected au niveau de l'instance on s'adresse généralement à this. Toutefois ceci est possible :

```
public class Fille extends org.demos.generation_x.Mere {  
    public void modifieValeur( Fille autre) {  
        autre.valeur = this.valeur ; // normal mais rarissime  
    }  
}
```

- Plus probant, car respectant l'encapsulation, est le cas de méthodes d'instance protected. Il s'agit de méthodes exposant un "mécanisme interne" à des sous-classes. S'agissant d'un mécanisme propre à l'instance les codes des sous-classes accèdent à ces méthodes au travers de this. Par exemple une sous-classe de Observable va invoquer this.setChanged().

Un exemple intéressant : la classe com.google.common.collect.AbstractIterator<T> définit une méthode protected abstract T computeNext() qui devra être implantée par toute sous-classe concrète. Cette méthode sera appelée (sur l'instance courante!) par les méthodes de la super-classe! De plus elle aura l'obligation d'appeler elle-même (toujours sur l'instance courante!) la méthode concrète de la super-classe protected final T endOfData()!

On remarquera qu'une méthode peut passer du statut de "mécanisme interne" au statut de "service". C'est le cas de la méthode clone() de Object. *A priori* on ne peut pas cloner un objet à partir d'un autre code que celui de la classe courante... sauf si la classe de cet objet spécialise clone() en la rendant publique (et en invoquant clone sur sa super-classe -et en implantant l'interface Cloneable-).

Les éléments static relèvent de modalités différentes :

- Une méthode protected static fournit un service réservé au package courant et à l'ensemble des sous-classes de la classe courante.
- Un champ protected static est considéré comme une erreur de *design* favorisant des effets de bord incontrôlés (sauf s'il s'agit d'une donnée immuable et final).

Formations JAVA sur ce sujet

Pour approfondir ces notions (encapsulation, contrôles de responsabilité, etc.) vous pouvez suivre les cours suivant:

- [USL275 - Java : Programmation pour Développeur Confirmé](#)
- [UJD110 - Programmation pour Développeur Débutant](#)
- Nos formations Java longues (standards ou spécifiques).
- Nos cours "java avancé" adaptés à la demande

A lire également : [Fiche pratique : JAVA SE - Encapsulation et Bean](#)

Bernard AMADE

© Demos