Préparation Orale - Conceptes et Langages Orientés Objets

Ce document est destinée a servire de repère dans les révisions des sujets d'examen du cours *Concepts et Langages Orientés Objets - CLOO*.

Chaque thème requis a l'examen est abordée de manière seule puis ouvert vers la généralité du cours.

Enumérations

Définition d'énumération

Une énumération est un ensemble prédéfini de constantes. Par exemple:

- mois dans l'année: janvier, février,...
- état d'une porte: ouvert, fermé, vérouillé
- couleurs aux échecs
- couleurs de cartes
- ...

Selon les langages ses constantes sont représenté par le système de différentes manières, en *C* par exemple ses constantes sont enfait des entiers, cela peut nôtament causer des erreurs de nomage, alors qu'en *java* il s'agit d'un type a part entière.

Les énumérations sont particulièrement utiles pour tester des égalités prédéfinit ou des états. Elles sont généralement utilisées en combinaisons avec des `switch case` puisqu'elles possèdent des valeurs fixe.

Exemple d'énumération en *java*

Déclaration d'une énumération:

```
public enum DoorState {
    OPEN, CLOSED, LOCKED
}
```

Utilisation d'une énumération:

```
import static DoorState.*;

DoorState open( DoorState current ) {
    if( current == CLOSED )
        return OPEN;
    else if( current == LOCKED )
        throw new IllegalStateException("Cannot open a locked door");
```

```
        else throw new IllegaleStateException("Door is already open !");
    }
```

Utilisation en combinaison avec un `switch case`:

```
import static DoorState.*;

DoorState open( DoorState current ) {
    switch(current) {
        case CLOSED:
            return OPEN;
        case LOCKED:
            throw new IllegalStateException("Cannot open a locked door");
        case OPEN:
            throw new IllegaleStateException("Door is already open !");
    }
}
```

Généricité et varience

Héritage et aggrégation

Héritage et Polymorphisme

Héritage: redéfinition des méthodes

Identité et Egalité

Lambda Expressions

Mutabilité et immutabilité

Polymorphisme et interfaces

Principe de substitution de Liskov

Références et construction

Références et passage d'argument

Références, méthodes et mutabilité

Visibilité et Contrôle d'accès

Principe et mots-clefs de la visibilité et du Contrôle d'accès

La visibilité est une propriété contextuel, c'est-à-dire qu'elle est dépendante de l'endroit ou on se trouve. Il est possible de voir un certain attribut à un endroit mais pas a un autre.

Le contrôle d'accès est une notion fortement liée à la visibilité puisqu'un attribut qui ne peut être vu ne peut être directement accéder par un utilisateur.

Il existe différents mots-clefs lié à la visibilité et à l'accès en orienté objet, plus spécifiquement en *java*:

- `public`
- `protected`
- `private`
- `default`

Ces mot-clefs sont contexte dépendant, dans le cas d'un attribut de class (méthode, constructeur, données):

- Un atttibut `public` peut-être vu et manipulé depuis l'extérieur de la class depuis n'importe quel contexte.
- Un attribut `protected` ne peut-être vu et manipulé qu'à l'intérieur du package dans lequel est défini sa classe, à l'exception d'une sous-classe de la classe de l'attribut.
- Un attribut `private` ne peut-être vu et manipulé uniquement depuis l'intérieur de sa classe.
- un attribut spécifié `default` revient à la même chose que de ne pas mettre de mots-clef d'accès. Le comportement est alors le même que `protected`.

Dans le cas d'une classe et d'une interface:

- Une classe `public` peut-être vu et manipulé depuis n'importe quel context d'où elle as été préaleblement défini ou importer.
- Une classe ne peut avoir l'attribut `protected` à l'execption d'une classe déclàré dans une classe.
- Une classe `private` ne peut-être vu et manipulé uniquement a l'intérieur du contexte où elle as été définie.
- Une classe spécifié `default` a le même comportement que si aucun mot-clefs d'accès est défini. Dans ce cas, la classe ne peut-être accéder que depuis le package où elle as été déclarée.

Dans certain cas il est généralement désirable qu'un utilisateurs ne puisse directement accéder à certains attribut d'une classe, afin de cacher l'implémentation et la rendre plus abstraite. Dans ce cas, on peut spécifié l'attribut comme *privé*, `private`, et on définie certaines méthodes pour y accéder et si besoin le modifié. Ces méthodes d'accès sont appellées *getter*, en français *accesseur*, et les méthodes de modifications sont appellées *setter*, en français *modifieur*.

En général, il est désirable que ces *getter* et *setter* retournent des *intefaces* pour cacher les détails d'implémentation, par exemple si l'on décide de changer le type de retour d'un *getter*. Si on change d'un `int` à un `long`, ci on récupère souvent la valeur on peut se retrouver a devoir changer plusieurs centaines lignes de code pour peut de choses. Alors que si, par defaut, ils retournaient une *interface*, il suffirait de modifier certaines méthode de la dites *interface*.