Java
Notions complémentaires

LICENCE 3 Sciences pour l'Ingénieur Parcours Informatique

Sommaire

- Les annotation
- Les classes abstraites
- Les interfaces
- Les classes internes
- Programmation fonctionnelle et lambdas

Les annotations

Les annotations

- Les annotations sont des marqueurs qui permettent d'ajouter des métadonnées aux classes, méthodes, attributs, paramètres ou paquets
- But :
  - ajouter des informations qui pourront être exploitées par le programme
  - améliorer la lisibilité de votre code
  - optimiser le contrôle des erreur potentielles
- Intégrées dans Java 5
- Une annotation est un type (comme une classe ou une interface). Elle n'est pas instanciée.

Les annotations

- Exemple :

```
public class Voiture extends Vehicule{
    @Override
    public String toString() {
        return "une voiture";
    }
}
```

- **@override :** signale au compilateur qu'une méthode est une redéfinition d'une méthode déclarée dans une classe parente
  - Vérification supplémentaire du compilateur

Les annotations

- **@deprecated** : Permet de générer des warnings afin d'informer les autres développeurs que quelque chose (une classe, une méthode…) a été dépréciée et ne devrait plus être utilisée
- **@SuppressWarnings** : Permet de forcer le compilateur à ne plus émettre d'avertissement à la compilation dans certains cas.

Les annotations

- Certaines annotations déclarent des attributs (paramètres) :
  - `@SuppressWarnings(value ={"deprecation", "unused"})`
  - Ignore les Warnings de type *deprecation* et *unused*
- Les attributs d'une annotation peuvent-être uniquement :
  - un type primitif
  - une chaine de caractères
  - une référence de classe
  - une annotation
  - une énumération
  - un tableau à une dimension d'un de ces types

Les annotations

- Créer son annotation personnalisée :

```
public @interface MonAnnotation{
     String auteur() default "Moi";
     String date();
     String commentaires();
}
```

- Réutiliser notre annotation :

```
@MonAnnotation(auteur="Guéniot", date="19 oct 2022")
public void maMethode(){
}
```

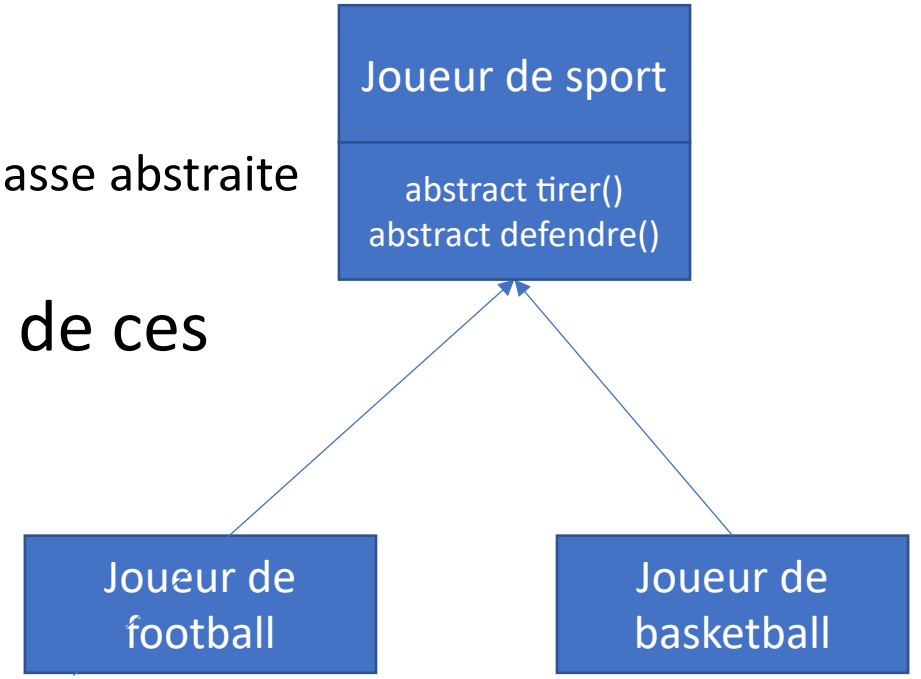
Les classes abstraites

Les classes abstraites

- Une classe abstraite est une classe qu'on ne souhaite pas pouvoir instancier (pas de new)
- Une classe abstraite peut ou non contenir des méthodes abstraites
- Une méthode abstraite ne possède pas de définition (pas de code...)
- Une classe dérivée d'une classe abstraite ne redéfinissant pas toutes les méthodes abstraites est elle-même abstraite
- Une méthode statique ne peut pas être abstraite car on ne peut redéfinir une méthode statique

Les classes abstraites

- Quel intérêt ?
  - A définir un comportement commun à plusieurs classes sans rien figer
  - A définir un concept
  - Exemple : un joueur de sport collectif
    - Comment définir des méthodes tirer() ou défendre() ?
    - On les défini dans la classe mère sous la forme d'une classe abstraite (sans code)
- Une classe doit-être abstraite si au moins une de ces méthodes est abstraite
- Une classe abstraite peut contenir des méthodes non-abstraites



Les classes abstraites : exemple

```
public abstract class JoueurDeSport{
     public abstract void tirer();
     public abstract void defendre();
}

public class JoueurDeFoot extends JoueurDeSport{
     // Redéfinition
     public void tirer(){
          // Tirer avec le pied !
     }
}
```

Les interfaces

Les interfaces

- Principe : déclarer un ensemble de méthodes permettant de définir un comportement, un ensemble de services visibles depuis l'extérieur
  - Ex : spécifier une API
- Les interfaces peuvent intégrer des déclaration de constantes
- Classe où toutes les méthodes sont abstraites. Le modificateur `abstract` est optionnel.
- Toutes les déclarations de méthodes sont par défaut publiques
- Les déclarations de variables sont implicitement des constantes : `public static final`
- Une classe indique qu'elle implémente une ou plusieurs interfaces en utilisant le mot clé `implements`

Les interfaces : exemple

```
public interface Carnivore {
        void manger(Animal animal);
}
public interface Herbivore {
        void manger(Vegetal vegetal);
}
public class Humain extends Animal implements Carnivore, Herbivore {
        @Override
        public void manger(Animal poulet) { // ... }
        @Override
        public void manger(Vegetal salade) { // ... }
}
```

Héritage d'interface

- Une interface peut hériter d’autres interfaces. Contrairement aux classes qui ne peuvent avoir qu’une classe parente, une interface peut avoir autant d’interfaces parentes que nécessaire. Pour déclarer un héritage, on utilise le mot-clé extends.
- ```
  public interface Omnivore extends Carnivore,
                                    Herbivore { }
  ```
  ```

Classe abstraite ou interface ?

- Classe abstraite :
  - Factoriser du code
  - Certaines méthodes sont implémentées, d'autres non
- Interface :
  - Définir un contrat de service
  - Aucune méthode n'est implémentée

Alors pourquoi ne pas utiliser juste des classes abstraites ?
Parce qu'une classe ne peut hériter que d'une seule classe (abstraite ou non) mais peut implémenter plusieurs interfaces

Les classes internes

Les classes internes

- On utilise une classe interne static pour cacher une classe dans une autre, sans avoir besoin de fournir à la classe interne une référence à un objet de sa classe englobante
- Si la classe interne est déclarée avec le modificateur static elle a accès à toutes les variables static de la classe englobante même, les static private
- Elle n’a pas par contre accès aux variables d’instances

Les classes internes

```
public class ClasseExterne{
    private int index = 0;

    class ClasseInterne{
        public void afficher(){
            System.out.println(index);
        }
    }
}
```

Les classes internes anonymes

- Utiliser une classe interne… sans déclarer de nom de classe
- Utilisables comme variable d'instance

Les classes internes anonymes

```
public static void main(){
    Voiture voiture = new Voiture(){
        void rouler(){
            ...
        }
    };
    voiture.rouler();
}
```

Programmation fonctionnelle et lambdas

Programmation fonctionnelle

- Java est un langage impératif. Il permet cependant d'utiliser des paradigmes de la programmation fonctionnelle.
- Programmation fonctionnelle :
  - Le code peut-être exécuté par de appels successifs de fonctions.
  - Va favoriser la représentation d'une application comme un appel chaîné de fonctions pouvant prendre elles-mêmes des fonctions en paramètre.
  - Avantages : code facilement testable, plus précis et concis.
  - Inconvénients : les données (variables) ne sont pas modifiables. Ne convient pas à toutes les tâches

Les fonctions anonymes : les lambdas

- Une lambda est une fonction anonyme, c'et à dire une fonction qui est déclarée sans être associée à un nom.
- Ajoutées dans Java 8
- Syntaxe : `(paramètres) -> { corps }`
- Exemple : `(int a, int b) -> { return a + b; }`
- Equivalent de la fonction :

```
public static int somme(int a, int b) {
    return a + b;
}
```

Les fonctions anonymes : les lambdas

- Exemple : forEach

```
Collection<String> collection = new ArrayList<>();
collection.add("un");
collection.add("deux"); collection.add("trois");
collection.forEach(e -> System.out.println(e));
```

- Exemple : Tri de tableau

```
List<Integer> liste = new ArrayList<>();
liste.add(1);
liste.add(2);
liste.add(3);
liste.add(4);
// trie la liste en plaçant en premier les nombres pairs
liste.sort((e1, e2) -> (e1 % 2) - (e2 % 2));
// [2, 4, 1, 3]
System.out.println(liste);
```

Les interfaces fonctionnelles

- Java ne support pas la notion de fonction -> **méthode ≠ fonction**
- Introduites dans Java 8, les interfaces fonctionnelles permettent de définir une interface disposant d'une seule méthode abstraite
- Une interface fonctionnelle donne juste la possibilité d'utiliser une notion, sans coder son comportement
- On peut préciser ce comportement avec l'annotation `@FunctionalInterface`
- Exemple :

```
@FunctionalInterface
public interface OperationSimple {
   int calculer(int i);
}
```

Les interfaces fonctionnelles

- Pour appeler une interface fonctionnelle et lui affecter un comportement, nous faisons appel à une lambda :

```
@FunctionalInterface
public interface OperationSimple {
    int calculer(int x);
}

OperationSimple op = (int x) -> x * x;
int calcul = op.calculer(5);
```