

Rappels de POO

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



UFR IEEA
Formations en
Informatique de
Lille 1

Langage à objets (pur)

- “Tout est objet”
- “Un programme est un regroupement d’objets qui se disent quoi faire par envois de messages”
- “Chaque objet a sa propre mémoire constituée d’autres objets”
- “Chaque objet a un type”
- “Tous les objets d’un type donné peuvent recevoir le même type de messages”

Alan Kay

Classe

Une classe décrit un **type**.

Elle précise les **attributs** et les **méthodes** de ses **instances**, ainsi que les **constructeurs**.

Objet

Un objet = **instance** d'une classe. Conforme au "*modèle*".

objet = identité + état + comportement

Les paquetages

- permettent de regrouper des types (cohérence logique ou fonctionnelle)
- déclaration implicite : `package monpaquetage;`
- importation de paquetage : `import monpaquetage;`
- compilation, exécution : `CLASSPATH`

↔ revoir le “TP 4” de POO !

Exceptions

- anticiper les portions de code susceptibles de générer une erreur
- fournir un mécanisme permettant d'apporter une réponse en cas de situation qui ne correspond pas au fonctionnement normal du programme
- laisser au programmeur le choix d'apporter ou non une réponse à la situation anormale
- les exceptions sont prises en charge par le *gestionnaire d'exceptions*
- une exception est un objet de type `Exception`

Capture

- instruction `try ... catch`
- veiller à l'ordre de capture
- instruction `finally` : code exécuté qu'il y ait eu exception ou non

```
private Map<Participant,Integer> tableResultat;  
public int enregistreResultat(Participant p, int res) {  
    try {  
        // ((...tableResultat.get(p)) vaut null si p est inconnu  
        int ancRes = this.tableResultat.get(p);  
        if (res > ancRes) {  
            this.tableResultat.put(p,res);  
        }  
    } catch(NullPointerException npe) {  
        // le participant est inconnu  
        this.tableResultat.put(p,res);  
    }  
}
```

Levée d'exception

- déclaration par `throws`
- levée par `throw`

```
public int meilleurResultat(Participant p)
    throws ParticipantInconnuException {
    if (this.tableResultat.containsKey(p)) {
        return this.tableResultat.get(p);
    } else {
        throw new ParticipantInconnuException(p+" est inconnu.");
    }
}
```

Les interfaces Java

Une interface

- est un type,
- impose des signatures de méthodes (sans code),
- peut être **implémentée** par des classes
↪ dans ce cas la classe a **obligation** de respecter le contrat de l'interface.

Les interfaces permettent la **généricité** et le **polymorphisme**.

Polymorphisme

- avoir plusieurs points de vue sur un même objet.
- pouvoir utiliser un objet dans plusieurs contextes.

Polymorphisme = “plusieurs formes”
Objet → **plusieurs types**

Polymorphisme

- avoir plusieurs points de vue sur un même objet.
- pouvoir utiliser un objet dans plusieurs contextes.

Polymorphisme = “plusieurs formes”
Objet \rightarrow **plusieurs types**

Distinguer classe de l'objet et type d'une référence sur l'objet.

```
TypeRef ref = new UneClasse();
```

Doivent être “compatibles” mais pas identiques.

Exemple : épreuves sportives.

- différentes épreuves, chacune avec leurs caractéristiques : *4×100-4 nages, perche, slalom géant, marathon, etc.*
- elles appartiennent à des disciplines : *athlétisme, natation, ski, etc.*
- elles correspondent à des types d'épreuves : *course, saut, lancer, etc.*
- elles peuvent ou non être *olympiques*

QuatreCentsQuatreNages

Disque

Tremplin

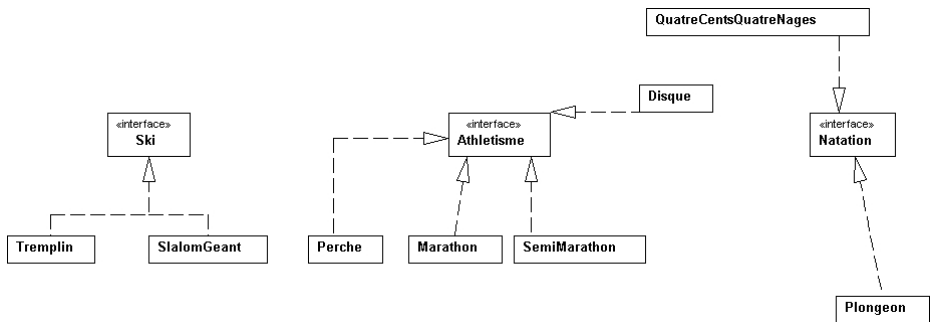
SlalomGeant

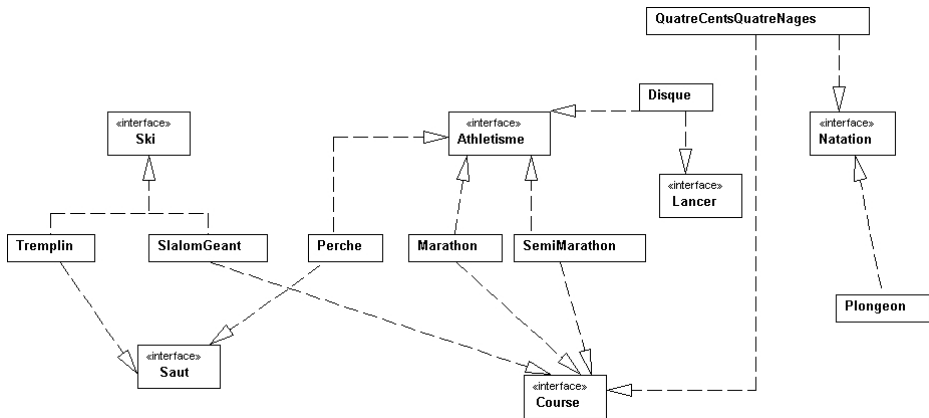
Perche

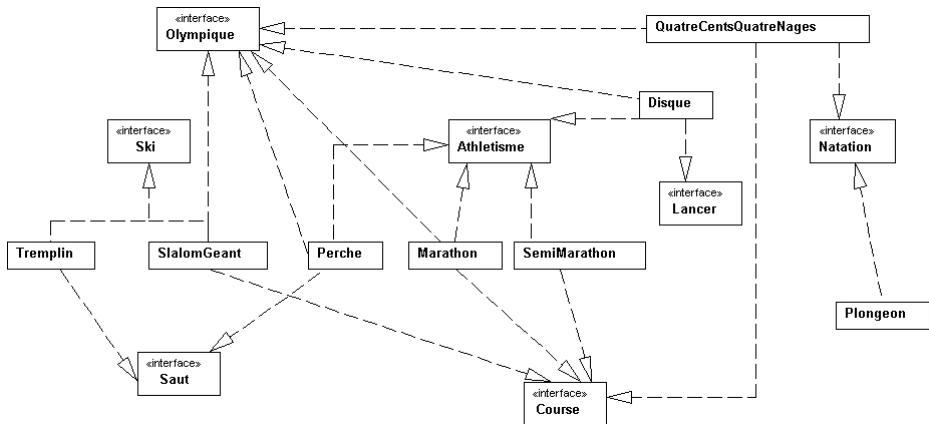
Marathon

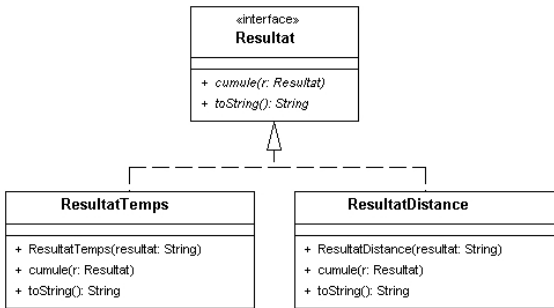
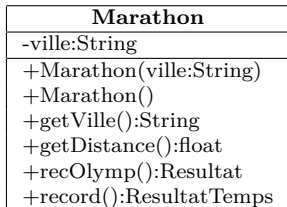
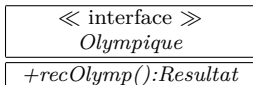
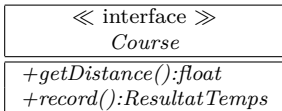
SemiMarathon

Plongeon









- Un objet est du type de sa classe **et** de toutes les interfaces implémentées par sa classe.
une instance de `Marathon` est à la fois de type : `Marathon`, `Athlétisme`, `Course`, `Olympique`, `Object`
- Toute référence de l'un de ces types peut donc pointer sur cet objet.

- Un objet est du type de sa classe **et** de toutes les interfaces implémentées par sa classe.
une instance de `Marathon` est à la fois de type : `Marathon`, `Athlétisme`, `Course`, `Olympique`, `Object`
- Toute référence de l'un de ces types peut donc pointer sur cet objet.

```
Marathon m = new Marathon();  
Course c = m;  
Olympique o = m;  
Natation n = m;           // interdit
```

Upcast et Downcast

UpCast

Changer vers une classe moins spécifique (toujours possible vers Object) : **généralisation**, opération “sûre”.

Upcast et Downcast

UpCast

Changer vers une classe moins spécifique (toujours possible vers Object) : **généralisation**, opération “sûre”.

```
Marathon m = new Marathon();  
Course c = m;           // UPCAST  
Natation n = m;       // illégal : détecté à la compilation
```

Upcast et Downcast

UpCast

Changer vers une classe moins spécifique (toujours possible vers Object) : **généralisation**, opération “sûre”.

```
Marathon m = new Marathon();  
Course c = m;           // UPCAST  
Natation n = m;        // illégal : détecté à la compilation
```

DownCast

Changer vers une classe plus spécifique : **spécialisation**, opération “à risque”.

Upcast et Downcast

UpCast

Changer vers une classe moins spécifique (toujours possible vers Object) : **généralisation**, opération “sûre”.

```
Marathon m = new Marathon();
Course c = m; // UPCAST
Natation n = m; // illégal : détecté à la compilation
```

DownCast

Changer vers une classe plus spécifique : **spécialisation**, opération “à risque”.

```
Marathon m = new Marathon();
Course c = m;
Marathon m2 = (Marathon) c; // DOWNCAST
SlalomGeant sg = (SlalomGeant) c; // DOWNCAST illégal ⇒ compile !
```

- Le type de la référence détermine les envois de message autorisés.
- La classe de l'objet détermine le traitement réalisé.

- Le type de la référence détermine les envois de message autorisés.
- La classe de l'objet détermine le traitement réalisé.

```
Marathon m = new Marathon();
System.out.println(m.getVille());
System.out.println(m.getRecord());
Olympique ol = m;
System.out.println(ol.recOlymp()); // code de Marathon exécuté
System.out.println(ol.getVille()); // refusé (à la compilation)
ol = new Perche();
System.out.println(ol.recOlymp()); // code de Perche exécuté
```

Une référence de type interface ne propose qu'un accès restreint à l'objet.

Principe Ouvert Fermé

OCP

Un code doit être ouvert à l'extension et fermé à la modification.

On doit pouvoir ajouter des éléments sans perturber l'existant.

Principe Ouvert Fermé

OCP

Un code doit être ouvert à l'extension et fermé à la modification.

On doit pouvoir ajouter des éléments sans perturber l'existant.

```
public void afficheRecord(Olympique olymp) {  
    S.o.p(olymp.recOlymp());  
}
```

```
Marathon m = new Marathon("Paris");  
truc.afficheRecord(m);
```

Principe Ouvert Fermé

OCP

Un code doit être ouvert à l'extension et fermé à la modification.

On doit pouvoir ajouter des éléments sans perturber l'existant.

```
public void afficheRecord(Olympique olymp) {
    S.o.p(olymp.recOlymp());
}
```

```
Marathon m = new Marathon("Paris");
truc.afficheRecord(m);
```

- Si on **ajoute** la classe `CentDixMètresHaies` qui implémente `Course`, `Athlétisme` et `Olympique`.
- On peut sans **rien modifier** avoir :


```
truc.afficheRecord(new CentDixMètresHaies());
```

enum

(java \geq 1.5)

enum permet la définition de types énumérés

```
public enum Saison { hiver, printemps, ete, automne;}
```

Référence des valeurs du type énuméré :

```
Saison s = Saison.hiver;
```

- En fait, création d'une classe avec un nombre prédéfini et fixe d'instances.
- Les valeurs du type sont donc des instances de la "classe enum".
↪ Saison est une classe qui a (et n'aura) que 4 instances, Saison.printemps est l'une des instances de Saison.

Méthodes fournies

Pour un type énuméré **E** créé, on dispose des méthodes.

Méthodes d'instances :

- **name():String** retourne la chaîne de caractères correspondant au nom de *this* (sans le nom du type).
- **ordinal():int** retourne l'indice de *this* dans l'ordre de déclaration du type (à partir de 0).

Méthodes de classe (static) :

- **static valueOf(v:String):E** retourne, si elle existe, l'instance dont la référence (sans le nom de type) correspond à la chaîne *v* (réciproque de **name()**).
- **static values():E[]** retourne le tableau des valeurs du type dans leur ordre de déclaration

Exploitation

```
public enum Saison { hiver, printemps, ete, automne;}
```

```
// ailleurs
```

```
public class Test {
    public void suivante(String nomSaison) {
        Saison s = Saison.valueOf(nomSaison);
        int indice = s.ordinal();
        Saison suivante = Saison.values()[((indice+1)%(Saison.values().length))];
        System.out.println("apres "+nomSaison+" vient "+suivante.name());
    }
}
```

```
public static void main(String[] args) {
    Test t = new Test();
    if (args.length > 0) {
        t.suivante(args[0]);
    }
    else {
        t.suivante("hiver");
    }
}
}
```

Que se passe-t-il ?

Le compilateur crée la classe (à peu près) :

```
public final class Saison extends java.lang.Enum {
    private static int cpt =0;
    private String name;
    private int index;
    private Saison(String theName){
        this.name = theName;
        this.index = cpt++;
    }
    public static final Saison hiver = new Saison("hiver");
    public static final Saison printemps = new Saison("printemps");
    public static final Saison ete = new Saison("ete");
    public static final Saison automne = new Saison("automne");
    public String name() { return this.name; }
    public int ordinal () { return this.index; }
    public static Saison[] values() {
        return { Saison.hiver, Saison.printemps, Saison.ete, Saison.automne };
    }
    public static Saison valueOf(String s) { // à peu près
        if (s.equals("hiver") { return Saison.hiver; }
        else if (s.equals("printemps") { return Saison.printemps; }
        // idem pour ete et automne...
    }
}
```

- Constructeur **privé**.

Logistique à faire soi même en java ≤ 1.4

Ce sont des classes...

On peut donc ajouter des attributs, méthodes, constructeurs...

```
public enum Coin {
    penny(1), nickel(5), dime(10), quarter(25);    // constantes

    private final int value;                        //attribut

    private Coin(int value) {                      // constructeur
        this.value = value;
    }

    public int getValue() { return this.value; }    // méthode
}

// usage
for(Coin c : Coin.values()) {
    System.out.println(c.name()+" vaut "+c.getValue());
}
```

Tables de hachage

Tables d'associations *clé* \leftrightarrow *valeur*.

- le type Map
 - \hookrightarrow HashMap, TreeMap
 - méthodes : put, get, remove, containsKey, containsValue, keySet, values, entrySet, etc.
 - pas une Collection : donc pas itérable
- problème sur les clés : **méthodes** hashCode, equals

Utilisation

Dans les `HashMap`

- le “*hashCode*¹” de la clé est utilisé pour retrouver rapidement la clé (sans parcourir toute la structure).
↪ par défaut la valeur de la référence.
- la méthode `equals()` est utilisée pour gérer les collisions (2 clés avec même *hashCode*)

donc pour que 2 objets soient considérés comme des clés identiques, **il faut** :

- qu'ils produisent le **même** `hashCode`
- qu'ils soient **égaux** du point de vue de `equals`

⇒ définir des fonctions `hashCode()` (aïe !) et `equals(Object o)` adaptées pour les clés des `HashMap` (et donc valeurs des `HashSet`)

¹`int` obtenu à partir de l'objet par une fonction de hachage