Bases de la Programmation Orientée Objet / Java

DUT / Module M213

Le paradigme objet

Définitions d'un paradigme

➔ **Petit Larousse illustré**

« *Ensemble des formes fléchies d'un mot pris comme modèle (déclinaison ou conjugaison)* (Ling.)»

« *Ensemble des unités qui peuvent être substituées les unes aux autres dans un contexte donné* (Ling.)»

« ***Modèle théorique de pensée*** *qui oriente la recherche et la réflexion scientifique»*

Paradigmes de programmation (1)

➔ **Wikipedia**

« Style fondamental de programmation informatique qui traite de la manière dont les solutions aux problèmes doivent être formulées dans un langage de programmation»

- *Un paradigme de programmation fournit (et détermine) la vue qu'a le développeur de l'exécution de son programme*
- *Par exemple, en programmation orientée objet, les développeurs peuvent considérer le programme comme une collection d'objets en interaction*

Paradigmes de programmation (2)

➔ **Exemples (*source Wikipedia*)**

- Prog. impérative procédurale (C, Ada, Basic, Fortran)
- Prog. impérative fonctionnelle (Lisp, Scheme, Caml)
- **Prog. orientée objets** (Smalltalk, Eiffel, Java)
- Prog. orientée composants (OLE, JavaBeans)
- Prog. descriptive (XML, HTML)
- Prog. en logique (Prolog)
- Prog. orientée pile (Forth)
- Prog. synchrone (Esterel)
- …

Paradigme orienté objets

➔ **Position dans le cycle de vie**

- Assurer la continuité depuis la phase d'analyse (UML)
- Prolongement des concepts des S.G.B.D.
- Simplifier la phase de conception
- Encadrer la phase de programmation (Encapsulation)
- Faciliter l'intégration et la mise au point
- Améliorer la productivité (Réutilisabilité, extensibilité)
- Faciliter les phases de tests et d'intégration
- Faciliter la maintenance (Lisibilité, modularité)

Concepts sous-jacents

➔ **Niveau conception / programmation**

- instanciation d'objets à partir de classes
- l'encapsulation
- la relation de composition
- la relation d'héritage
- le polymorphisme
- la généricité
- la persistance

Conception orientée objets

➔ **Structuration logique**

- Toute application est une partition de **classes**
- Chaque classe est la description complète d'une entité fonctionnelle du problème à traiter
- Les entités sont identifiées en phase d'analyse (UML)
- Les **attributs** sont les propriétés de chaque entité
- Les **méthodes** sont les opérations (comportements) de chaque entité décrite
- Un diagramme de classes UML rassemble et relie entre elles toutes les classes du problème traité

Forte analogie avec les T.A.D (1)

➔ **Module M1103 / Langage C**

- Organisation hétérogène interne des données (**struct**)
- Accessibilité implicite externe à tous les champs
- Allocation dynamique des instances (fonction dédiée)
- Traitements décrite par fonctions publiques
- Nécessité de prévoir les moyens de destruction (**free**)

Exemple du T.A.D. Point

➔ **Module M113 / Langage C**

```
struct Spoint {
    double abscisse ;
    double ordonnee ;
};
typedef struct Spoint point;
point creerpoint(double uneAbs, double uneOrd);
double distance(point unA, point unB);
char* toString(point unA);
```

Forte analogie avec les T.A.D (2)

➔ **Module M213 / Langage Java**

- Le mot clé **class** remplace **struct**
- Introduction spécificateurs de portée (**private**/**public**)
- Masquage implicite de la définition de type (**typedef**)
- Pas d'accessibilité externe aux champs (**private**)
- Moyens standards de construction (**constructeurs**)
- Allocation dynamique des instances (**objets**)
- Consultation/modification des champs par **accesseurs**
- Encapsulation des fonctions (**méthodes**)

Exemple de la classe **Point** (1)

➔ **Module M213 / Langage Java**

```
public class Point
{
    private double abscisse;
    private double ordonnee;

    public Point (double uneAbs, double uneOrd)
   {      abscisse= uneAbs;
          ordonnee= uneOrd;
    }
```

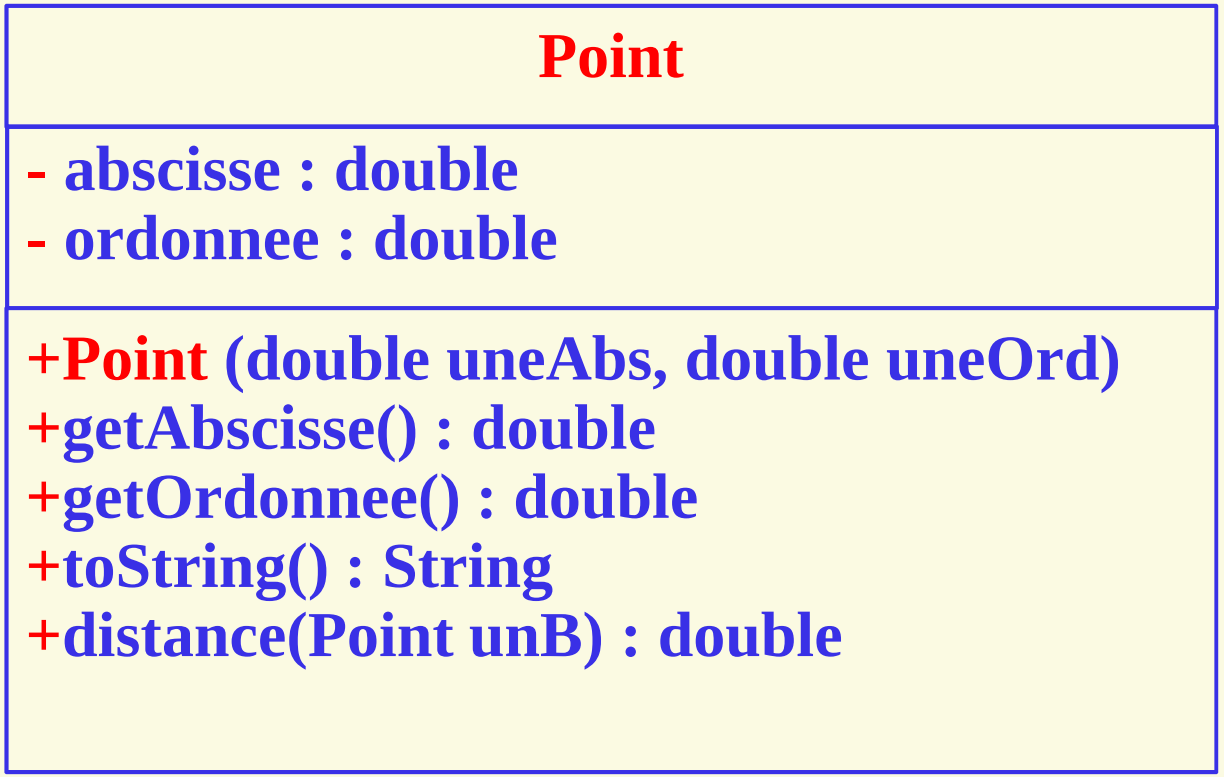
Exemple de la classe **Point** (2)

```
---
    public double getAbscisse() {return abscisse;}
    public double getOrdonnee() {return ordonnee;}

    public String toString()
    {
          return "("  + abscisse + " , "   + ordonnee + ")" ;
    }
---
```

Exemple de la classe **Point** (3)

```
---

    public double distance(Point unB)
   {
    double deltaX= abscisse - unB.abscisse;
    double deltaY= ordonnee  - unB.ordonnee;
        return Math.sqrt(deltaX*deltaX + deltaY*deltaY);
    }
}
```

UML


Point
- abscisse : double
- ordonnee : double
+Point (double uneAbs, double uneOrd)
+getAbscisse() : double
+getOrdonnee() : double
+toString() : String
+distance(Point unB) : double

Structuration d'un programme Java

➔ **Tout programme sera partitionné en classes**

- Chaque classe modélise une entité du problème traité
- Le terme **entité** est employé au sens OMGL
- Chaque classe décrit toutes les **propriétés** et toutes les **opérations**
- Tous les modèles sont **encapsulés** (visibilité limitée !)
- L'une des classes contient le point d'entrée (main)

Description d'une classe Java (1)

➔ **Bloc englobant de plus haut niveau**

- En tête l'identification de la classe (mot réservé : **class**)
- Précédé du spécificateur **public**
- Contient la totalité de la définition d'une classe
- Pas de compilation séparée (donc pas de prototypes !)
- Exemple de la classe **Point**

public class Point { --- }

Description d'une classe Java (2)

➔ Attributs

- Chaque propriété sera mémorisée dans un **attribut**
- Tous les attributs sont typés
- Factorisation syntaxique possible du type
- Tous les attributs sont privés (**private**)
- Exemple de la classe **Point**

private double abscisse;
private double ordonnee;

Description d'une classe Java (3)

➔ Constructeurs (1)

- **Moyen unique** de fabriquer des instances (**objets**)
- Possibilité de définir plusieurs constructeurs
- Chaque constructeur porte le nom de la classe
- Pas de mention du retour (ni type, ni void)
- Constructeur par défaut (sans paramètres)
- Constructeurs normaux (signatures différentes)
- Possibilité d'invoquer un constructeur dans le corps de la définition d'un autre constructeur

Description d'une classe Java (4)

➔ Constructeurs (2)

- Exemple pour un objet de la classe **Point**

```
public Point() {
    abscisse= 0.;
    ordonnee= 0.;
}
public Point(double x, double y) {
    abscisse= x;
    ordonnee= y;
}
```

Description d'une classe Java (5)

➔ Constructeurs (3)

- Mot clé **new** pour **invoquer** un constructeur
- Exemple pour un objet de la classe **Point**

```
Point p0= new Point();
Point p1= new Point(1.5, -2.7);
```

Description d'une classe Java (6)

➔ Accesseurs de consultation

- Restitution de la valeur courante d'un attribut
- Convention de nommage JavaBeans
- Exemple pour un objet de la classe **Point**

```
double x= p1.getAbscisse();
```

- Mise en œuvre de l'opérateur d'application **.**

Description d'une classe Java (7)

➔ Accesseurs de modification

- Modification externe de la valeur courante d'un attribut
- Définition optionnelle en fonction du besoin
- Exemple pour un objet de la classe **Point**

```
p1.setAbscisse(7.25);
```

- Mise en œuvre de l'opérateur d'application .

Structuration d'une classe Java (8)

➔ Opérations et comportements

- Chaque opération sera définie par une **méthode**
- La syntaxe des méthodes est analogue aux fonctions C
- Les méthodes sont donc paramétrées
- Possibilité de limiter
- Exécuter une méthode : l'appliquer à un objet support
- Exemple pour deux objets de la classe **Point**

```
Point p2= new Point(1., 0.5);
double d= p1.distance(p2);
```

Structuration d'une classe Java (9)

➔ **Cas particulier de la méthode toString**

- Analogue à la fonction de même nom pour les T.A.D.
- N'est pas à proprement parler une opération sémantique
- Indispensable pour les tests unitaires
- Fournit une représentation externe de tout objet, simple à comparer (message)
- Exemple pour un objet de la classe **Point**

```
String s= p1.toString();
```

Structuration d'une classe Java (10)

➔ **Cas particulier de la méthode main (1)**

- Point d'entrée (exécution) d'un programme Java
- Signature formelle unique et imposée
- Créer les objets applicatifs de plus haut niveau
- Exécute les comportements de plus haut niveau fonctionnel
- Définition nécessairement incluse dans une classe
- Plusieurs classes de la même application peuvent définir une méthode main

Exemple

```
public class Exemple
{
    public static void main (String[] args)
    {
        System.out.println("Hello world !");
     }

}
```

Structuration d'une classe Java (11)

➔ Cas particulier de la méthode main (2)

- Exemple pour la classe **Point** :

```
public static void main (String[] args) {
Point p0= new Point();
Point p1= new Point(0.5, 3.45);

    double d= p1.distance(p0);
    System.out.println ("Distance= "  +  d);
}
```

Le langage Java

➔ Langage objet semi compilé

- langage à structure de blocs (blocs imbriqués)
- chaque classe est un bloc
- chaque méthode est un sous bloc
- chaque structure de contrôle est un sous bloc
- portée des variables limitée au bloc de déclaration