

# Cours n° 4

## Objets, classes, interfaces et héritage

## Sommaire

### 1. Introduction à la programmation objet

1. Paradigmes de programmation
2. Concepts de l'orienté-objet

### 2. Classes et objets

1. Attributs et méthodes
2. Création, manipulation et destruction d'un objet
3. Dérivation de classes et de méthodes

### 3. Classes abstraites et interfaces

#### 1.1 PARADIGMES DE PROGRAMMATION

### Programmation fonctionnelle (Lisp, Scheme, ML)

#### Principes de programmation

Programme est une expression fonctionnelle (spécification de ce qui doit être calculé, pas de la méthode de calcul)

Exécution d'un programme est l'évaluation de l'expression

Pas de notion de variables, d'états, de séquence

Pas de gestion de la mémoire par le programmeur

Utilise la récursivité au lieu des itérations

```
(define fact (lambda (n) if (< n 2) 1 (* n (fact (- n 1)))))  
(fact 6)
```

#### Propriétés

Preuve mathématique du bon fonctionnement d'un programme,

Difficulté d'estimation de la complexité (explosion combinatoire)

#### 1.1 PARADIGMES DE PROGRAMMATION

### Programmation fonctionnelle par contraintes (Alice)

#### Principes de programmation

Satisfaction de contraintes entre variables,

Fonctions de contrainte définies sur des ensembles de cardinalité finis

Ensemble de variables et de domaines de définition associés

Soient trois variables entières X, Y et Z définies sur {1..9},  $X = 2Y + 3Z$

Solutions {(5, 1, 1), (8, 1, 2), (7, 2, 1)}

Planning, productique, étiquetage morpho-syntaxique, ...

#### Propriétés

Heuristiques de recherche,

Contrôle du risque d'explosion combinatoire

**Programmation impérative (Fortran)****Principes de programmation**

Modifications successives de l'état global (mémoire) de la machine par une suite de commandes,

- Détermination des données nécessaires au calcul,
- Association des emplacements mémoires aux données,
- Description des transformations à appliquer aux données

Instruction principale: affectation ( $x = 1$ ;  $x = x+1$ )

boucles: while, repeat

**Propriétés**

- Plus algorithmique, plus proche du comportement d'un ordinateur,
- Pas de preuve du bon fonctionnement,
- Effets de bord sur les variables du programme (visibilité)

**Programmation impérative procédurale (Pascal, C, Perl)****Principes de programmation**

Encapsulation des suites d'instruction exécutées à plusieurs reprises dans des procédures,

- Programme structurée en une suite d'appels de procédure,
- Limitation de la visibilité des variables associées à une procédure,
- Règles pour le passage de paramètres (en entrée et en sortie)

**Propriétés**

- Durée limitée d'exécution pour une procédure,
- Pas de sauvegarde (sauf exception) de l'état interne d'une procédure,
- Réduction des effets de bord sur les variables,
- Compilation séparée

**Programmation impérative orientée-objet (C++, Java)****Principes de programmation**

- Ensemble d'objets communiquant par l'intermédiaire de message,
- Objet : Encapsulation de variables et des procédures associées,
- Limitation de la visibilité des procédures associées (fonctions membres, méthodes),
- Interface de l'objet : procédures visibles (publiques) traitant les messages,
- Visibilité des variables par toutes les procédures associées à l'objet (sauf exception)

**Propriétés**

- Adaptation au développement en équipe
  - Spécifications préalable des interfaces,
  - Développement séparé de chaque objet (écriture et test),
- Méthodologies de conception orienté-objet

**Classes****Regroupement d'objets utilisant les mêmes méthodes et la même structure de données**

Définition d'un moule d'objet (instance d'une classe est un objet)

**Attributs d'une classe**

- Ensemble des types formant la structure de données,
- Instanciation de la classe -> Instanciation des attributs
  - Création de variables internes de l'objet
- Un attribut peut être lui-même une autre classe

**Etat de l'objet**

Ensemble des variables internes de l'objet

**Objet****Un objet est un exemplaire ou une instance d'une classe**

Tout objet appartient à une classe

Chaque objet possède un nom unique

Un objet vit indépendamment des autres objets

Etat (individuel) qui tient compte de l'effet des opérations

Un objet peut changer son état propre

**Messages et méthodes****Résultat de l'envoi d'un message à un objet**

Connaissance de l'état de l'objet,

Modification de l'état de l'objet,

Comportement indépendant de l'état de l'objet

**Méthodes**

Suite d'opération permettant de répondre à un message,

Déclenchement d'une méthode publique à chaque message,

La mise en œuvre d'une méthode est cachée à l'appelant

**Héritage****Attributs et méthodes communes à plusieurs classes**

Création d'une super classe à partir des caractéristiques communes

**Définition d'une hiérarchie de classe**

Classe = Super classe + attributs particuliers + méthodes particulières,

Héritage par la classe des caractéristiques de la super-classe,

Classe Object (super classe dont hérite toutes les autres)

arborescence de définition (en Java, pas en C++)

**Héritage multiple**

Possibilité d'hériter de plusieurs super-classes (en C++, pas en Java).

**Simplification de la réutilisation****Syntaxe de déclaration (1/4)**

**[modificateur\_de\_classe] class** ClassName **[extends** superClass]

{ // attributs, méthodes et constructeurs }

Déclaration visible des classes du même paquetage (regroupement logique d'un groupe de classes)

**Modificateurs de classe**

abstract Définition d'une classe abstraite (au moins une méthode abstraite)

final Classe non dérivable (héritage de cette classe interdit)

public Classe utilisable en dehors du paquetage

Utilisation des classes du paquetage P (import P.\*;)

## Syntaxe de déclaration (2/4) - Attributs

**[modificateur\_d'attribut]** TypeName AttrName;

Variable instanciée correspondante accessible à toutes les classes du paquetage

### Modificateurs d'attributs

public	Accessible en dehors de la classe (déconseillé)
private	Accessible uniquement dans la classe (conseillé)
protected	Accessible aux classes dérivées des classes du paquetage
final	Variable non modifiable (après initialisation)
static	Variable instanciée commune à toutes les instances de la classe (Variable de classe)

## Déclaration de la classe Date

```
package cours04;                // librairie courante

import util.Keyboard;          // librairies utilisées

/** Création et gestion de dates (jour, mois, année)
 * @author montacie */

public class Date {
    //attributs de classe
    private int jour, mois, année;
```

## Syntaxe de déclaration (3/4) - Méthodes

**[modificateur\_de\_méthode]** [TypeName] MethName (paramètres) {}

Méthode accessible par défaut à toutes les classes du paquetage

### Modificateurs de méthode

public	Accessible en dehors de la classe (interface de la classe)
private	Accessible uniquement dans la classe
protected	Accessible aux classes dérivées des classes du paquetage
static	Pas de Modification de l'état de l'objet (hors variables de classe)
final	Non redéfinissables dans une classe dérivée
synchronized	Traite les messages d'une seule tâche à la fois

## Méthodes de comparaison et d'entrées-sorties

```
/** Comparaison entre deux dates
 * @param d Date comparée */

public boolean CompareTo (Date d) {
    if (année != d.année) return true;
    if (mois != d.mois) return true;
    if (jour != d.jour) return true;
    else return false; }

/** Affichage de la date */

public void Afficher() {
    System.out.println(jour+" "+mois+" "+année); }

/** Lecture au clavier de la date */

private void Lire() { // méthode lecture d'une date
    jour = Keyboard.getInt("Entrez le jour");
    mois = Keyboard.getInt("Entrez le mois");
    année = Keyboard.getInt("Entrez l'année");}
```

## Méthode de calcul

```

/** Calcul de la date du lendemain */
public void Incrementer () {
    // Pas de taille indiquée dans la déclaration des tableaux
    int lmois[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    // prise en compte des années bissextiles
    if (((année % 4) == 0) && ((année % 400) != 0)) lmois[1]++;
    jour++;
    if (jour > lmois[mois-1]) {
        jour = 1; mois++;
        if (mois == 13) {année++; mois = 1;}
    }
}

} // fin de la déclaration de la classe date

```

## Syntaxe de déclaration (4/4) - Constructeur

**public** ClassName (paramètres) {corps de la méthode constructeur}

## Méthode publique

Accessible en dehors de la classe (sauf pour les classes privées)

## Nom de la méthode imposée

Nom de la classe

## Possibilité de constructeurs multiples

Polymorphisme des méthodes

## Pas de paramètre de retour

## Constructeurs

```

/** Création et initialisation d'une nouvelle instance de Date
 * @param j jour
 * @param m mois
 * @param a année
 */
public Date(int j, int m, int a) {
    jour = j; mois = m; année = a;
}

/** Création et lecture clavier d'une nouvelle instance de Date
 */
public Date() {Lire();}

```

## Création d'un objet

ObjName = **new** ClassName[(Paramètres du constructeur)];

## Instanciation d'une classe en 4 phases

Création et initialisation à zéro de variables correspondant aux attributs,

Initialisation explicite des attributs,

Choix et appel d'une méthode constructeur de l'objet,

Renvoi d'une référence sur l'objet

## Invocation d'une méthode et accès à l'état d'un objet

ObjName.MethName(Paramètres de la méthode); **ou**

ClassName.MethName(Paramètres de la méthode); cas d'une méthode statique

### Mode de passage des paramètres

Par référence pour les objets,

Par valeur pour les types primitifs

[ObjName.]AttrName

Nom de l'objet optionnel en cas d'accès par une méthode de la classe courante

## Utilisation de la classe Date

```
package cours04;

public class testDate {
    /** point d'entrée d'un exécutable */
    public static void main(String[] args) {
        // Instanciation de deux objets de type date
        Date today = new Date();
        Date dfin = new Date();

        today.Afficher(); dfin.Afficher();
        do { today.Incrementer(); today.Afficher(); }
        while (today.CompareTo(dfin));
    }
}
```

06/10 Cours 5 s'arrête ici

20/10 Cours 6 commence ici

## Méthodes de la classe originelle « Object »

<b>String toString()</b>	Retourne le nom de l'objet
<b>Class getClass()</b>	Retourne un descripteur de la classe
<b>int hashCode()</b>	Retourne une valeur entière correspondant à un code de hachage de l'objet
<b>Object clone(Object)</b>	Retourne une copie de l'objet
<b>boolean equals(Object)</b>	Retourne vrai si les deux objets sont égaux
<b>void finalize()</b>	Appelée juste avant la destruction de l'objet (instant décidé par la JVM)

## Principes de l'héritage en langage Java

### Héritage simple

Une classe ne dérive que d'une seule super-classe,

Attributs et méthodes de la super-classe inclus dans la classe dérivée,

Une classe dérivée n'accède pas aux attributs et méthode privées de sa super-classe

### Instanciation d'une classe dérivée en 6 phases

Création et initialisation à zéro de variables correspondant à tous les attributs,

Initialisation explicite des attributs de la super-classe,

Appel d'une méthode constructeur de la super-classe,

Initialisation explicite des attributs supplémentaires de la classe dérivée

Appel d'une méthode constructeur de la classe dérivée,

Renvoi d'une référence sur l'objet

## Principes

## Redéfinition des modificateurs de la classe d'origine

Dans un sens plus restrictif (public -> protected -> none -> private)

## Redéfinition de méthodes et d'attributs de la super-classe

Définition dans la classe dérivée d'une méthode ou d'un attribut portant le même nom qu'une méthode ou un attribut de la super-classe,

Interdit pour les méthodes et les attributs avec le modificateur « final »,

Masquage de la méthode ou de l'attribut d'origine,

Déconseillée pour les attributs,

Accès par le mot-clé « super » aux méthodes et aux constructeurs de la super-classe

super. MethName(Paramètres de la méthode);

super(Paramètres du constructeur);

## Déclaration d'une classe dérivée

```
package cours04;

import java.util.Calendar;

/** Création et gestion améliorées de dates
 * (seconde, minute, heure, jour, mois, année)
 */
public class Date2 extends Date {
    private int seconde, minute, heure;
```

## Redéfinition des constructeurs

```
public Date2(int j, int m, int a) {
    super(j, m, a); // appel du constructeur de Date
    heure = Calendar.HOUR_OF_DAY;
    minute = Calendar.MINUTE;
    seconde = Calendar.SECOND;
}

public Date2() {
    super(0, 0, 0); // sinon appel automatique du
                  // constructeur vide de Date
    heure = Calendar.HOUR_OF_DAY;
    minute = Calendar.MINUTE;
    seconde = Calendar.SECOND;
}
```

## Redéfinition des méthodes

```
/** Redéfinition - Affichage de la date et de l'heure */
public void Afficher() {
    super.Afficher();
    System.out.println(heure+" "+minute+" "+seconde);
}

/** Affichage de l'heure */
public void Afficher2() {
    System.out.println(heure+" "+minute+" "+seconde);
}

// fin de la définition de la classe Date2
```

## Utilisation de la classe dérivée

```
package cours04;

public class testDate2 {

    public static void main(String[] args) {
        Date2 heure = new Date2();
        heure.Afficher2();// Affichage de l'heure

        Date2 today = new Date2(24, 10, 2003);
        today.Afficher();// Affichage du jour et de l'heure
    }
}
```

## Définitions

## Propriétés

- Une classe abstraite doit contenir au moins une méthode abstraite
  - méthode avec le modificateur abstract (en général sans bloc d'instructions)
- Interdiction de l'instanciation d'une classe abstraite (erreur à la compilation)
- Création indispensable de sous-classes pour une utilisation
  - Sous-classes abstraites ou concrètes
- Classe concrète
  - Implémentation de toutes les méthodes abstraites

## Choix d'une base de méthodes communes à toute une hiérarchie de classes

- Garantie d'une redéfinition spécifique des méthodes pour chaque classe concrète,
- Amélioration par rapport à la redéfinition (non obligatoire) de méthodes

## Définition d'une classe abstraite

```
/** création et gestion d'une figure géométrique */
public abstract class FigureGeometrique {
    protected String couleur = "noir"; // couleur par défaut
    /** création d'une nouvelle instance de FigureGeometrique
     * @param co couleur de la figure */
    public FigureGeometrique (String co) {
        couleur = co; // initialisation de la couleur }
    /** calcul du périmètre d'une figure géométrique
     * @return périmètre */
    public abstract float périmètre();
    /** calcul de la surface d'une figure géométrique
     * @return surface */
    public abstract float surface();
} // fin de définition de la classe
```

## Définition de classes concrètes (1/2)

```
/** création et gestion d'un carré rouge */
public class CarreRouge extends FigureGeometrique {
    protected float côté = 0;
    /** Création d'une nouvelle instance de CarreRouge */
    public CarreRouge(float x) {
        super("rouge"); côté = x; }
    /** calcul du périmètre d'un carré rouge
     * @return périmètre */
    public float périmètre() { return 4*côté; }
    /** calcul de la surface d'un carré rouge
     * @return surface */
    public float surface() { return côté*côté; }
} // fin de définition
```



## Définition de classes concrètes (2/2)

```

/** création et gestion d'un rectangle bleu */
public class RectangleBleu extends FigureGeometrique {
    private float grand_côté = 0, petit_côté = 0;
    /** Création d'une nouvelle instance de RectangleBleu
     * @param x petit côté du rectangle
     * @param y grand côté du rectangle */
    public RectangleBleu(float x, float y) {
        super("bleu"); grand_côté = x; petit_côté = y; }
    /** calcul du périmètre d'un rectangle bleu
     * @return périmètre */
    public float périmètre() {return 2*(grand_côté+petit_côté);}
    /** calcul de la surface d'un rectangle bleu
     * @return surface */
    public float surface() {return grand_côté*petit_côté; } }

```

## Définition

## Propriétés

Une interface correspond à une classe abstraite où toutes les méthodes sont abstraites.

Une classe peut implémenter une ou plusieurs interfaces tout en héritant d'une classe.

Une interface peut hériter d'une autre interface.

## Intérêt

Simulation de l'héritage multiple

Pas d'attributs modifiables,

Peu de simplification de code (méthodes à réécrire)

Aide à la conception

Nombre important d'interfaces disponibles

## Définition d'une interface

```

/** gestion de la couleur d'une forme géométrique */
public interface Coloriable {
    /** liste des couleurs autorisées */
    static String[] couleurs =
    {"rouge", "orange", "jaune", "vert", "bleu", "violet"};

    /** modification de la couleur d'une forme géométrique
     * @param co nouvelle couleur
     */
    public abstract void changerCouleur(String co);
} // fin de définition

```

## Implémentation d'une interface

```

/** création et gestion de carrés multicolores */
public class CarreMulticolore extends CarreRouge implements
Coloriable {
    /** Création d'une nouvelle instance de CarreMulticolore
     * @param x côté du carré
     * @param co couleur du carré */
    public CarreMulticolore(float x, String co) {
        super(x); this.changerCouleur(co); }
    /** modification de la couleur d'un carré
     * @param co nouvelle couleur */
    public void changerCouleur(String co) {
        int i;
        for (i = 0; i < couleurs.length; i++)
            // est-ce une couleur autorisée ?
            if (co.equals(couleurs[i]) == true) break;
        if (i < couleurs.length) couleur = co;
    }
} // fin de définition

```