Université Montpellier-II
UFR des Sciences - Département Informatique
Licence Informatique -

Ingénierie Logicielle - Concepts et Outils de la modélisation et du développement de logiciel par et pour la réutilisation.

Programmation par classes en Java

Notes de cours
Christophe Dony

1 Définition d'une classe en Java

1.1 Rappel

Classe : entité

- représentant un ensemble d'objets ayant la même représentation mémoire (structure) et les mêmes comportements,
- définissant cette structure
- définissant et détenant ces comportements
- capable de générer des instances.

1.2 Syntaxe de définition

- *public* ou *private // portée*
- *abstract // impossibilité à être instanciée*
- *final // impossibilité à être dérivée (Voir Plus Loin)*
- **class "nom de la classe"**

public class - *extends "nom_de_la_super-classe" // (VoirPlusLoin)* toto n toto

- *implements "nom_d'interface"* // (VPL)*
- **{ "corps de la classe" }**

Exemple :

```
1  public class Point {
2      private int x;
3      private int y;
5      public int gety() {return y;}
7      public Point setx(int i) {
```

```
8         x = i;
9         return this;}
10 }
```

1.3 Les membres d'une classe

Membre : élément définissant les propriétés structurelles ou fonctionnelles des instances d'une classe : variable d'instance, variable de classe, constructeur, méthode d'instance, méthode de classe, classe imbriquée, destructeur.

variable d'instance :

(ou *attribut* ou *variable membre*), définit une propriété structurelle ou relationnelle (par opposition à fonctionnelle ou comportementale) des instance d'une classe - sert à la matérialisation des relations de composition et d'association mises en évidence pendant l'étape de conception.

champs : chaque instance d'une classe possède une valeur propre de chaque variable d'instance définie par sa classe ; il est parfois dit que cette valeur est stockée dans un "champs" de l'instance. Cette valeur peut être fixée au moment de l'instantiation ; elle peut être modifiée (dans le respect des règles de visibilité de l'attribut) tout au long de la vie d'une instance.

1.4 Déclarations d'une variable d'instance

- *private — protected — public — package // portée*

- *static // il s'agit d'une variable de classe (VPL),*

- *final // la variable ne pourra être affectée qu'une seule fois*

- *transcient // la variable ne doit pas être prise en compte lorsqu'une instance de la classe doit être sauvée sur disque (VPL serialization).*

- *volatile // empêche certaines optimisations du compilateur*

- **type et nom de la variable**

Exemple :

```
1 public class Point {
2   private double x, y;
3   . . . }

5 public class Rectangle {
6   public Point origine;
7   public float longueur, largeur;
8   . . . }
```

1.5 La portée (visibilité) des membres

Portée d'un identificateur : Zone du texte d'un programme, relativement à l'emplacement de sa déclaration, ou un identificateur est visible.

Specifier	class	subclass	package	world
private	X			
protected	X	X*	X	
public	X	X	X	X
package	X		X	

`package` est le qualifieur par défaut. Un membre `x` de visibilité `package` est visible dans tout le package où `x` a été déclaré.

2 Création des objets

Un objet est instance d'une classe.

2.1 Instantiation

Instantiation : nom donné à l'opération de création d'un objet à partir du modèle qu'est une classe.

Objet : instance d'une classe, entité informatique, individuelle, anonyme, repérée par une adresse unique, et constituée d'un ensemble de champs contenant des valeurs.

Rappels : Un objet possède autant de champs qu'il y a de variables d'instance définies dans sa classe. Les valeurs de ses champs représentent l'état courant d'un objet.

2.2 Instantiation en Java

Opérateur new : l'instantiation utilise l'opérateur *new*[1]

Exemples :

```
1 new Point();
2 new Rectangle();
```

l'opérateur *new* crée pour le nouvel objet autant d'emplacements memoire qu'il y a de variables d'instance dans sa classe et rend l'adresse de l'objet (ne pas confondre l'objet et la variable dans lequel on stocke éventuellement son adresse).

Si on souhaite concerver l'adresse d'un objet il faut la stocker quelque part, dans une variable, une liste ou un tableau.

Exemples :

```
1 Point p1 = new Point();
2 Rectangle rect = new Rectangle();
3 Point[] tab = new Point[10]; // un tableau de 10 points
4 tab[1] = new Point(); // dans le second emplacement duquel on stocke un nouveau Point
```

Référence en Java (sur le modèle de *Lisp*, *Scheme* ou *Smalltalk*) : Identificateur dans lequel est stockée l'adresse d'un objet et pour lequel est opéré un déréférencement automatique et systématique lors de l'accès . L'adresse n'est jamais accesible en tant que telle. L'utilisation obligatoire de références assure que les objets sont toujours passés *par référence*, donc sans recopie. (La comparaison entre les références de Java et celles de C++ est plus complexe et dépasse le cadre de ce cours).

[1] pas d'objets automatiques comme en C++.

```
Point* pp = new Point(); // un pointeur C++
Pont& rp = *pp; //une référence C++ obtenue par déréférencement explicite.
```

3 Les méthodes

Méthode : fonction ou procédure nommée, définie au sein d'une classe et définissant une propriété fonctionnelle des instances de la classe. Elle décrit comment une instance de cette classe va réagir à la réception du message correspondant à son nom.

3.1 Définition des méthodes en Java

- *private* | *protected* | *public* | *package* // visibilité

- *static* // méthode de classe,

- *abstract* // méthode abstraite ou virtuelle pure (VoirPL),

- *final* // méthode ne pouvant être redéfinie sur une sous-classe (ne pouvant être spécialisée),

- *synchronized* // méthode à exécuter en exclusion mutuelle, permet d'assurer la synchronization de différentes exécutions de la méthode en cas d'utilisation de processus concurrents *threads*).

- **"Type de la valeur rendue"**

- **"nom de la méthode"**

- **("liste des paramètres")**

- *throws "liste des exceptions signalées"*

3.2 Exemple

Exemple pour la classe `Point` :

```
    public double getx() {return x;}

    public Point setx(double i) {
        x = i;
        return this;}
```

3.3 Envoi de message (ou appel de méthode)

Syntaxe Java : *receveur.selecteur(arg1, ..., argn)*

Envoi de message : demande à un objet, dit le *receveur* d'exécuter une méthode définie dans sa classe (à voir une seconde version de cette définition prenant en compte le mécanisme d'hétitage) par une méthode de nom *selecteur*.

Exemples :

```
Rectangle rect = new Rectangle();
rect.move (20,20);
rect.area();
```

Receveur : au sein d'une méthode `M` de nom `m` d'une classe `C`, objet, instance de `C`, auquel a été envoyé le message `m` ayant entraîné l'exécution de `M`.

Attention, le même opérateur "." est utilisé pour l'accès à un attribut, qui n'est pas un envoi de message.

```
1 int area = rect.height * rect.width
```

En Java, le compilateur vérifie (statiquement donc) qu'une telle méthode existe pour tout envoi de message, mais il ne peut pas toujours déterminer laquelle, qui ne sera connue qu'à l'exécution (voir "liaison dynamique"). C'est de cette incertitude que vient la puissance des langages à objets en terme d'extensibilité et de réutilisabilité.

3.4 Les identificateurs accessibles au sein des méthodes

Les identificateurs accessibles au sein d'une méthode M sont :

- les paramètres de M,
- les variables locales de M,
- les variables d'instance de l'objet receveur O,
- la pseudo-variable `this`,
- les variables de classe de la classe C de O (celle ou est définie M)

Rappels :

portée (scope) : ensemble des régions d'un programme où un identificateur peut être référencé.

Extent (durée de vie) : Intervalle de temps pendant lequel on peut faire référence à un identificateur.

Paramètres et locales : portée lexicale, durée de vie dynamique.

Variables d'instances et de classe : portée dépendante de la visibilité déclarée (public, protected, private), durée de vie celle des objets.

3.5 Initialisation des objets

Les champs d'un objet O nouvellement créés sont tout d'abord initialisées avec les valeur par défaut du type de la variable correspondante.

exemple : Pour un Point, x = 0.0 et y = 0.0

Ensuite le constructeur de la classe de O ayant le nombre de paramètre correspondant à l'expression d'instantiation est exécuté.

3.6 Les constructeurs

constructeur : méthode particulière, portant le même nom que la classe et automatiquement invoquée juste après la création d'une instance.

Un constructeur peut avoir un nombre quelconque de paramètres.

Un constructeur sans paramètre est conseillé sur toute classe.

Un paramètre peut avoir le même nom qu'une variable d'instance, auquel cas il la masque en R-position.

L' **exécution d'un constructeur** débute par une initialisation éventuelle et automatique des attribus de l'objet si des valeurs ont été données à la déclaration. Elle se poursuit avec l'exécution des instructions du constructeur.

Exemple :

```
public class Point {
    double x = 0.0;
    double y;

    public Point() {
        x = 1.0;
        y = 2.0;}

    public Point(double i, double j){
        x = i;
        y = j;} ...}
```

```
class Rectangle{
    Point origin = new Point(0,0);
    Point corner;

    Rectangle(Point p1, p2) {
        origin = p1;
        corner = p2;} ...}

new Rectangle();
new Rectangle(new Point(), newPoint(3,4));
```

3.7 Destruction des objets

3.7.1 Ramasse-miette

Java dispose d'un ramasse-miette (garbage collector) de type *mark and sweep)* dans sa version *sun-jdk*. Donc, l'espace mémoire occupé par tout objet qui n'est plus référencé est automatiquement récupéré au bout d'un certain temps.

Il peut néanmoins être nécessaire de déréfencer certains objets ou de libérer les ressources non gérées par le GC (par exemple fermer un fichier).

Exemple : pour tout objet référencé, directement ou indirectement, dans une variable globale devenue inutile :

```
GlobalVariable = null
```

3.7.2 La méthode *finalize*

Il est possible, pour toute classe, d'écrire une méthode *finalize*, automatiquement invoquée lorsqu'un objet va être recupéré par le GC (Equivalent des destructeurs C++).

```
protected void finalize() throws Throwable {
  ...
  super.finalize();}
```

3.8 Types

Java est un langage statiquement typé : à tout identificateur et toute expression d'un programme est associé un type, dit type statique, dans le texte du programme.

Les types Java sont soit prédéfinis soit définis par une classe ou par une interface (toute classe et toute interface définissent un type).

3.8.1 Type primitifs

byte short int long float double char boolean

3.8.2 Type reférence

Un type dit "référence" est défini par une classe ou une interface. Toute classe et toute interface définit un type.

```
1 interface I { ... }
2 class A { ... }
3 A a = ... ;
4 I i = ... ;
```

Une variable de type référence contient l'adresse d'un objet mais ne rend pas l'adresse accessible.

Tout accès à une variable d'un type référence opére un "déférencement automatique".

Les "références" ont été popularisées par *Lisp* puis par *Smalltalk* et se sont imposées en programmation par objet car elles permettent le passage d'argument par valeur de toute structure (donc d'objets) sans copie de la structure et évitent également les erreurs liées à la manipulation explicite des adresses.

Seul C++ propose à la fois les types "pointeur" (T*) ET "référence" (T&).

4 liaison dynamique, unicité des noms, extensibilité

4.1 Liaison dynamique

Liaison nom d'opération - fonction : l'exécution d'un programme nécessite que soit établie une correspondance entre un nom de méthode dans le texte du programme et une méthode effective définie sur une classe.

Liaison dynamique : liaison réalisée à l'exécution du programme.

liaison statique : liaison réalisée au moment de la compilation (édition de liens).

Avec les langages à objet, la liaison est dynamique (sauf pour les méthodes statique et sauf optimisation du compilateur quand c'est possible) et l'interprétation d'un envoi de message est réalisée dynamiquement selon l'algorithme suivant.

4.2 Receveur d'un envoi de message

Dans une envoi de message "`a.m(...)`", a est appelé le receveur, m est appelé le sélecteur.

Dans toute exécution de méthode, l'objet qui a reçu le message ayant provoqué l'invocation de la méthode courante est accessible au sein de cette méthode via l'identificateur `this`.

`this` est un paramètre formel implicite de toute méthode.

```
class Stack {
    ...
    public int pop(){
        if this.isEmpty() ...}
    ...}
```

4.3 Interprétation de l'envoi de message, version 1

Interpretation1 : sans pré-compilation de l'envoi de message, ni prise en compte de l'héritage ni du contrôle des types des paramètres et en supposant qu'il n'y a qu'une seule méthode de nom *sel* par classe.

```
rec.sel(args)

    C := class (receveur)
    M := searchMethod (C, sel)
    Si (M différent de nil) alors apply (M, rec, args)
                  sinon erreur("MessageNotUnderstood", rec, sel)
```

4.4 Unicité des noms

Opération générique : entité abstraite représentant un ensemble d'opérations concrètes, de même nom, sur différents domaines de valeurs.

Exemple : l'opération générique *multiplier* notée *.

Restriction d'une opération générique O : une opération concrète de même nom que l'application générique.

Exemple : les diverses opérations de multiplication (entiers, réels, matrices, etc).

Unicité des noms d'opérations : possibilité de donner, dans un programme, le même nom externe à plusieurs restrictions d'une opération générique[2].

L'unicité des noms permet de décomposer une opération générique (par exemple la multiplication) en un ensemble d'opérations concrètes (la multiplication des entiers, des flottants, des matrices, etc) toutes invocables via le même nom externe.

Résolution de l'ambiguité : Lors d'un envoi de message, l'interpréteur dynamiquement, utilise le type dynamique du receveur (et éventuellement des autres arguments) pour déterminer quelle fonction doit être invoquée. Avec le typage statique, le compilateur a pu préparer le terrain pour accélérer l

4.5 surcharge, liaison dynamique et extensibilité

La surcharge permet au programmeur de s'abstraire du type de l'objet auquel il veut appliquer une opération générique.

```
new Rectangle().toString()
new Point(2,3).toString()
```

[2]Initialement, on pouvait parler de surcharge des noms au sens général du terme dans lequel surcharge signifie qu'un même nom est utilisé plusieurs fois pour désigner des choses différentes. Dans le contexte de la programmation par objet, Il est devenu utile de distinguer le cas où deux opérations génériques différentes portent le même nom, cas que l'on qualifie de surcharge, du cas où n restrictions d'une même opérations générique portent également le même nom (qui est donc de facto surchargé), cas pour lequel on s'interdira néanmoins de parler de surcharge pour qu'il n'y ait pas de confusion possible avec le cas précédent.

La surcharge permet l'écriture de fonctions applicables à différents types de données et applicables également à de futurs types de données.

Exemple de la méthode *toString* et de son utilisation dans la méthode *println*.

Synthèse. La méthode *println* fonctionne correctement, sans modification et sans recompilation avec des types de données (e.g. Point) qui n'existaient pas lorsqu'elle a été écrite. Pourquoi ? Parce que :

1) La méthode de fabrication d'une chaîne de caractère a le même nom pour chaque type de donnée.

2) La liaison est dynamique.

5 Variables et Méthodes de classe

5.1 Variables de classe ("statiques")

variable de classe : variables représentant,

- des propriétés dont la valeur est partagée par toutes les instances d'une même classe.
  Exemples : (1) la classe des carrés et la variable nbCotés (une constante), (2) la classe des francais et la variable president de la republique.
- des propriétés de la classe.
  La classe X et la liste des méthodes qu'elle possède.

Elles sont appelées variables de classe en Smalltalk. Le mot-clé *static* vient de C++ et fait que l'on appelle aussi ces variables, variables statiques.

Les variables de classe sont des membres des classes au même titre que les variables d'instance. A ce titre elles peuvent posséder les caractérisques des membres de classes (*public*, *protected*, etc).

Là où elles sont visibles, les variables statiques peuvent être référencées, simplement par leur nom, au sein des méthodes de classe ou d'instance et par notation pointée via la classe ou via une instance.

```
1 class français {
2     public String nom, prénom, adresse;
3     int age, département;
4     public static président = JC;
5     ...}

7 Dupont = new Français();
8 Dupont.président;
```

5.2 Méthodes de classe (statiques)

Méthode de classe : méthode définissant une propriété fonctionnelle de la classe (et pas de ses instances).

5.3 Exemples de méthodes et de variables de classe

```
1 class CV {
2     private static int j1;
3     public static int j2;
4     public static void initCV() {j1 = 1; j2 = 2;}
5     //acces a une variable de classe dans une methode
```

```
    int use() {return (j1 + j2);}
}
```

```
class testClassVar {
  public static void main(String[] args){
      CV.initCV();
      CV o1 = new CV();
      System.out.println("acces a une CV via la classe: " + CV.j2);
      System.out.println("acces a une CV via un objet: " + o1.j2);
      System.out.println("acces a une CV dans une methode: " + o1.use());
  }
}
```

Exemple : Variable de classe `MAXVALUE` de la classe `Integer`.

```
Integer.Max_Value
```

Exemple : Méthode de classe `ParseInt` de la classe `Integer`

```
static int parseInt(String s)
        Parses the string argument as a signed decimal integer.
```

6 Pratique de l'association de l'agrégation et de la composition

- **Relation d'association** : relation spécifiant qu'un l'objet est associé, est sémantiquement connecté, à un **autre** objet indépendant.

- **Relations d'Agrégation et de Composition** : relations binaires entre un tout et ses parties, les parties étant des éléments constitutifs du tout.

- **Relation de Composition** (losange noir) : le mot composition est utilisé lorsque la partie n'a pas d'existance possible sans son tout.

6.1 Implanter les Associations

Une association (*emploie*) entre deux sortes d'objets peut être représentée de differentes manières :

- Une classe (`Emploi`) et deux attributs : `employeur` de type `Emploi` dans la classe `Compagnie` et `employé` de type `Emploi` dans la classe `Personne`.
- Deux attributs, attention à la gestion de l'inverse.

```
Classe Personne
    employeur : Compagnie

Classe Compagnie
    employés : collection de Personnes
```

- Un seul des deux attributs : pas de problème de gestion de l'inverse mais problème de navigabilité.

6.2 Implanter l'aggrégation et la composition

s'implémentent en Java en affectant une variable d'instance de l'agrégat à une instance d'une autre classe.

Voici un exemple :

```java
public class Pile
{
    Vector buffer;
    int taille;

    Pile(int i) {
        buffer = new Vector();
        taille = i;
    }

    public Pile empiler (Object i) {
    if (this.isFull() ... vector.addElement(i)
    }
```

La composition ne demande pas de travail supplémentaire, en effet la destruction du vecteur est automatique une fois que la pile le contenant a été détruite. En C++ il faut y faire attention.

7 Intermède : Quelques classes et particularités de Java

7.1 Retour sur les tableaux

Les tableaux en Java sont des choses étranges. Il existe une classe *Array* dont les tableaux ne sont pas des instances mais qui possède un ensemble de méthodes de classes permettant de donner des informations sur les tableaux. Les tableaux ne sont donc pas vraiment des objets mais on peut les utiliser comme tels (c'est magique) (vive Smalltalk) :

```java
int[] tab = new int[12];
Object a = tab;
int i = tab.length; // cablé ?
//int i = tab.getlength(); // ne marche pas
int j = Array.getLength(tab);
```

7.2 La classe Object et le type associé

La classe Object définit les méthodes applicables à tous les objets du système.

Tous les types référence (définis par des classes) sont des **sous-types** du type *Object*

Sous-type : Première intuition, un sous-type possède toutes les propriétés définies par ses sur-types.

7.3 Les classes "enveloppe" - Wrapper-class

Question : Comment ranger un entier dans un tableau d'objets sachant que les éléments des types prédéfinis ne sont pas des objets.

Réponse, en utilisant les classes *enveloppe* ou *wrapper*.

Exemple : Une instance de la classe enveloppe *Integer* permet d'encapsuler un entier. Un *Integer* est un objet.

```java
Object[] tab = new Object[10];
//tab[1] = 12; --> erreur
tab[1] = new Integer(12);
int i1 = ((Integer)tab[1]).intValue();
```

NB : Il est par ailleurs bien sûr possible de créer des tableaux d'entiers.

```java
int[] tab2 = new int[10];
```

```
2 tab2[1] = 12;
```

8 Les packages

8.1 Création

```
1 package graphics;
2 class Rectangle { ... }
3 class Circle { ... }
```

8.2 Usage

Pour utiliser une entité se trouvant dans un autre package, il est possible soit :

- de la référencer par son nom complet,

```
1 graphics.Rectangle myRect = new graphics.Rectangle();
```

- de l'importer :

```
1 import graphics.Rectangle;
2 Rectangle myRectangle = new Rectangle();
```

d'importer le **package** complet dans lequel elle se trouve :

```
1 import graphics.*;
2 Circle myCircle = new Circle();
3 Rectangle myRectangle = new Rectangle();
```

8.3 Gestion des fichiers

1. Décider d'un répertoire principal racine des applications Java (exemple /h-arcp/dony/obj/java)
2. Indiquer ce répertoire au compilateur et à l'interpréteur :

```
1 setenv CLASSPATH "/h-arcp/dony/obj/java"
```

3. Créer des sous-répertoires correspondant aux noms de vos packages,

```
1 mkdir graphics
2 cd graphics
```

4. Définir toutes les classes du package *graphics* dans des fichiers se trouvant dans le sous-répertoire *graphics*. Rappel : une seule classe publique par fichier.
5. Compiler le programme :

```
1 javac Rectangle.java
```

6. Exécuter le programme. Quel que soit le répertoire courant :

```
java graphics.Rectangle
```

9 Les exceptions - Première Présentation (sans héritage)

Le modèle de gestion des exceptions est inspiré de ceux de *Smalltalk* et de *C++*, eux-même inspirés ...

9.1 Signalement d'une exception prédéfinie

```
class Stack {
  int index = 0;
  Object[] buffer = new Object[10];

  void push(Object o) throws Exception {
    if (index == buffer.length)
      throw new Exception("La pile est pleine");
    //suite
  }
}
```

Un signalement d'exception provoque une recherche de traitant (handler en anglais) dans la pile d'exécution. L'instance signalée (*throw new Exception*) sera passée en argument à tous les handlers.

9.2 Définition d'un handler - instruction *try-catch*

On associe un traitant à un bloc avec l'instruction *try-catch*.

Il peut y avoir n clauses *catch*, une seule sera exécutée. Elle doivent être ordonnées en tenant compte de la hiérarchie des classes d'exceptions.

Notez la possibilité de rattraper différente exceptions avec une seule clause.

```
class testException {
  public static void main(String[] args){
    Stack testStack = new Stack();
    try {testStack.push(new Integer(1));}
    catch (Exception (e)
        { ... corps du traitant ... }
    finally { ... restaurations inconditionnelles }
    }
}
```

9.3 Traitement de l'exception

Dans le corps du traitant, on commence par éventuellement remettre le programme dans un état cohérent, ou afficher un message - on peut pour cela utiliser les information stockées dans l'instance de l'exception.

```
e.getMessage()

e.printStackTrace()
```

Puis il est possible :

- de signaler une nouvelle exception,

- de ne rien faire, l'exécution reprend après l'instruction *try-catch*,
- d'interrompre la méthode courante : instruction *return*
- d'interrompre l'exécution du programme : *System.exit()*.

9.4 Restaurations inconditionnelles - instruction *try-finally*

```
try { ... actions ...}
finally { ... restaurations ...}
```

Les instructions `restaurations` seront exécutées après l'éxécution des instructions `actions` quoi qu'il arrive durant leur exécution.

10 Classes emboitées

Il est possible de définir une classe EA (ou une interface) à l'intérieur d'une classe A. On parle globalement de *Nested classes*. On en trouve deux formes : *static* ou *inner*.

10.0.1 classes emboitées statiques

```
public class X{
   private long iv;
   ...
   public static class Y{
      public boolean m1, m2;
      ... }
}
```

La classe Y est définie à l'intérieur de la classe X, elle est un membre de cette classe, au même titre qu'une variable d'instance, elle peut donc être privée ou publique ...

La classe Y est statique. Elle peut en fait être utilisée comme n'importe quelle classe du système MAIS on ne peut y accéder qu'en fonction des accès définis par sa classe englobante et son nom, vu de l'extérieur est : $X.Y$.

A l'intérieur d'une méthode d'une classe emboitée statique, on n'a pas accès aux variables d'instances non publiques de la classe englobante.

10.0.2 Inner Classes

Version classique et claire des classes définies dans des classes.

```
public class CompteBanque{
   private long solde;
   private Action lastAction;

   private class Action {
      String act;
      private long soldeCourant;
      Action(String act) {
        this.act = act;
        soldeCourant = solde;}}

   public void deposer(long depot){
      solde = solde + depot;
```

```
14      lastAction = new Action("dépot");}
15  }
```

La classe *Action* est définie à l'intérieur de la classe *Compte*, elle est un membre de cette classe, au même titre qu'une variable d'instance, elle peut donc être privée ou publique ...

Dans une méthode d'une classe emboitée non statique, on a accès à tous les membres de la classes emboitante.

10.0.3 Local Inner Classes

Sont définies dans un bloc. Ce ne sont pas des membre d'une classe.

11 Classes Utilitaires Standard

11.1 Vector

Fournit les tableaux d'objets de taille variable.

11.2 Bitset

Vecteur de bits. Une version optimisée des vecteurs contenant des bits à true ou à false.

12 Classe paramétrées

Définition : Un classe paramétrée est un classe paramétrée par un type (défini par une interface ou par une classe).

13 Synthèse No 1

- Sur les langages à objets en général
  - La plupart des langages sont des langages à classes.
  - Les programmes sont architecturés autour des classes.
  - Les classes permettent d'implanter des types abstraits de données
  - La classe définit la structure de ses instances par un ensemble de variables (appelées variables d'instance, ou champs ou attributs ou slots) ainsi que les opérations qui peuvent leur être appliquées (ensemble de procédures ou fonctions appelées méthodes).
  - Une méthode peut être invoquée en envoyant un message à un objet.
  - Plusiers méthodes dans le système peuvent avoir le même nom externe (surcharge)
  - L'envoi de message résoud la surcharge des noms d'opérations.
- Sur Java plus particulièrement
  - Toutes les entités ont un type, toutes ne sont pas des objets, il existe des types primitifs et des types références.
  - Tout objet est instance d'une classe.
  - Toute opération d'un objet est invoquée par envoi de message.
  - Les méthodes d'instance (ou fonction membres) s'appliquent aux objets.
  - Les méthodes de classe (ou fonctions membres statiques) s'appliquent aux classes.

14 Premiers exemples de programmes.

14.1 Nouveaux types de données

Une application avec un ensemble de nouveaux types de données.

Ex : Gestion des documents d'une entreprise.

- Une classe pour chacun des types de données composant le système.

  Employé (nom, prenom, bureau, nombre-emprunts)
  Document (titre, reference, nombre-exemplaires, listeExemplaires)
  Exemplaire de document (document, numero, emplacement)
  Emplacement (type, lieu)
  Emprunt(exemplaire, emprunteur, date)

- Une classe représentant le gestionnaire de documents

  gestionnaire (employes, documents, emprunts)

- Une classe permettant de tester l'application dotée de la méthode *main*.

  Application (unGestionnaire).
  Cette classe peut éventuellement être la même que la classe gestionnaire.