

# PROGRAMMATION ORIENTEE OBJET AVEC JAVA

LICENCE MIAGE – L3  
Daniel Diaz

# Objectifs

- Acquérir les concepts fondamentaux de la POO
- Apprendre à les implémenter en Java
- Apprendre dès le départ le bon usage des concepts étudiés.

Support en PDF disponible ici:

<https://cours.univ-paris1.fr/fixe/UFR27-L3Miage-POO>

(si pas encore inscrit utiliser le compte visiteur et la clé)

Convention de Codage :

<http://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

Specs Java d'Oracle : <http://docs.oracle.com/javase/specs/jls/se8/html/>

# Programme du cours

Programme (incluant la mise en œuvre en Java):

- Notions de base: de classe, attributs, méthodes, encapsulation, instantiation, surcharge.
- Variables et méthodes de classe vs variable et méthodes d'instance.
- Notions avancées : héritage, polymorphisme, classe abstraite et interface.
- Rédiger et générer la documentation associée à une classe.
- Etude des classes basiques et standards de Java (chaines de caractères, mathématiques, flux d'entrées/sorties)
- Etude des structures collectives disponibles en Java.

# INTRODUCTION ET GENERALITES SUR JAVA

# Historique

1991: Naughton, Gosling et Sheridan (déçus par C++) créent un langage orienté objets, robuste et industriel nommé OAK.

1993: le web décolle grâce au logiciel Mosaic et SUN demande d'adapter OAK à l'Internet (le projet s'appellera Java).

1995 : sortie officielle de Java

1996: sortie du JDK 1.0.2 (Java Developer's Kit)

1996: la version 2.0 du navigateur Web Netscape accepte les applets Java

1998 : nouvelle dénomination : Plate-forme Java 2:

- édition standard
- édition entreprise

## **Edition Standard**

La plate-forme Java 2, édition standard version 1.2 (J2SE) désigne les spécifications abstraites et environnement de base liés à Java.

Cette plate-forme est implantée par Java 2 SDK édition standard version 1.2 (J2SDK), autrefois appelée JDK1.2.

Elle comprend J2RE et les outils de développement.

Java 2 Runtime Environment, standard édition version 1.2 (J2RE) est l'environnement minimum pour l'exécution des applications J2SE.

## **Edition Entreprise**

En plus de la version standard, des fonctionnalités liées aux serveurs des applications distribuées telles que :

- Enterprise Java Beans (EJB: beans côté serveur)
- Servlets, Portlets (applets côté serveur)
- Java Server Pages ou JSP (pages Web dynamiques avec Java)
- JDBC (connexion à des BD)
- Et bien d'autres...

1996: JDK1.0.2 (1ère version utilisable) : 211 classes et interfaces.

1997: JDK1.1 : 504classes et interfaces (ajout des classes internes, nouveau modèle pour la gestion des événements des GUI, sérialisation, JavaBeans, servlets, JDBC)

1998: J2SE 1.2 (Java 2 SDK, version 1.2 Standard Edition) : 1520 classes et interfaces (ajout de SWING pour les GUI, interfaces avec CORBA).

2002: J2SE 1.4 : 1991classes et interfaces.

2004: J2SE 5 : 3279 classes et interfaces (generic, for each, ...).

2006: Java SE 6 : 3793 classes et interfaces.

2009 Oracle rachète Sun

2011: Java SE 7: 4024 classes et interface. NB: OpenJDK (licence GNU GPL) vraiment fonctionnel, vraie alternative à OracleJDK.

2014: Java SE 8: 4240 classes et interfaces

## Propriétés de Java

Langage OO (pas de variables globales, pas de pointeurs explicites).

Gestion automatique de la mémoire dynamique: utilise un « garbage-collector » pour récupérer la mémoire inutilisée (pas de « libération » d'objet explicite).

Portable (syntaxe proche du langage C).

Utilisation d'une machine virtuelle (JVM) et d'un émulateur pour produire de exécutables indépendant de la machine.

Support du multi-tâches (grâce aux threads)

Adapté à la programmation Internet.

Assez sûr: fortement typé, pas de pointeurs « fous », vérifications au chargement des classes et durant l'exécution (débordements).

Java est fourni avec une bibliothèque très riche.

## Schéma de compilation et d'exécution

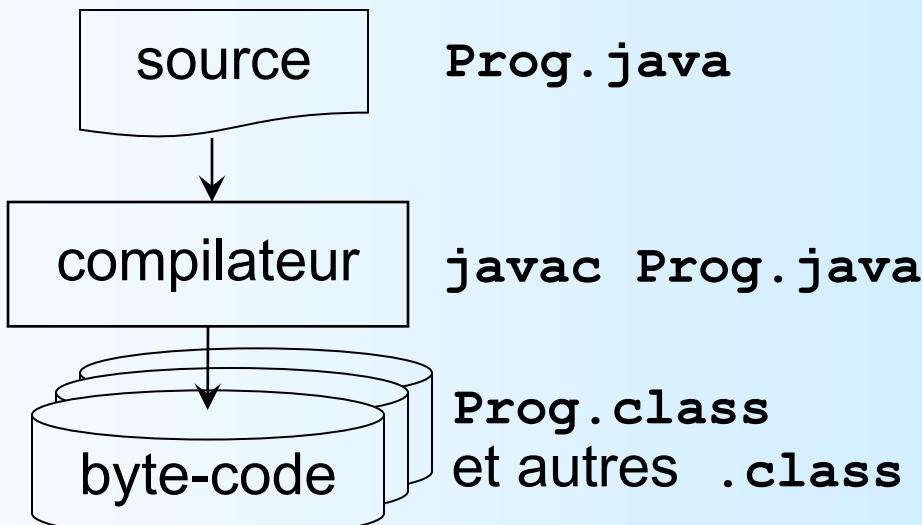
Le source Java (Prog.java) est compilé et donne lieu à un byte-code (Prog.class) pour la machine virtuelle de Java (JVM).

Ce byte-code ne peut être exécuté directement et nécessite un émulateur : c'est un programme qui lit le byte-code, le décode et l'exécute.

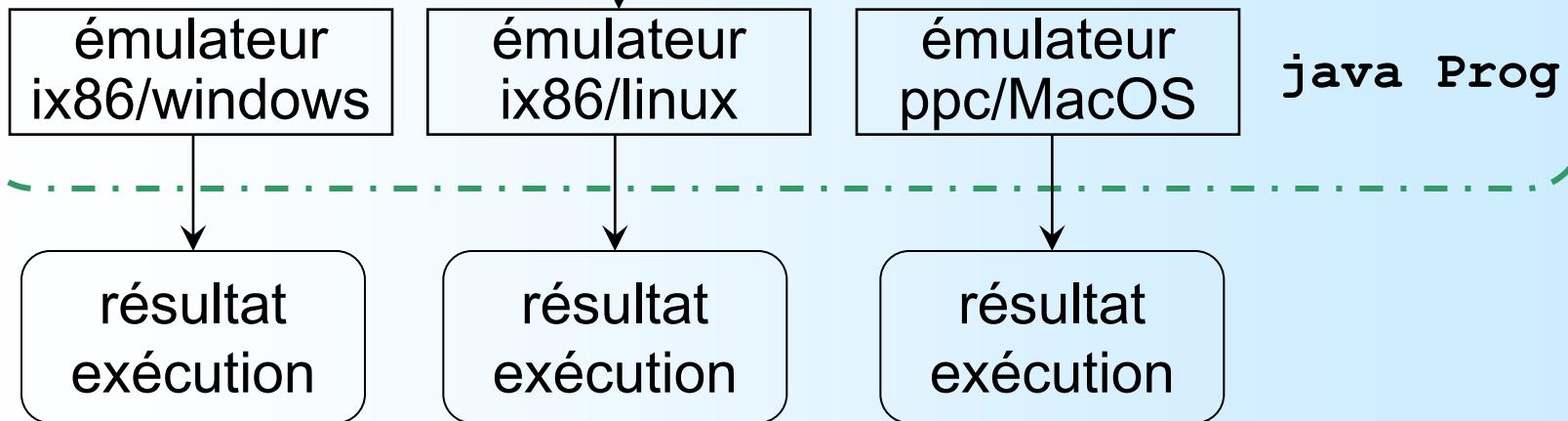
Le byte-code est indépendant de la machine et peut être exécuté sur n'importe quelle machine pourvu qu'un émulateur soit présent.

L'émulateur java est présent sur la plupart des architectures.

## *Compilation*



## *Exécution*



# **ELEMENTS DE BASE**

## **DU**

# **LANGUAGE JAVA**

# Structure générale

On ne détaille pas ici l'aspect OO: juste voir les éléments de base.

Syntaxe proche de celle du C, C++.

Un fichier source Java est constitué de *classes* (souvent 1 seule).

Une classe comprend la définition de variables (appelées *attributs*) et de fonctions (appelées *méthodes*).

Une application console doit contenir une méthode `main` définie à l'intérieur d'une classe portant le même nom que le fichier source (ATT: majuscules et minuscules sont différenciés en Java et doivent donc l'être dans les fichiers sources).

Déclaration de la fonction main:

```
public static void main(String [] args) {  
    ... corps du main ...  
}
```

# Définition de classes

Une classe se définit comme suit :

```
<protection> class <nom classe> {  
    ... définition d'attributs ou de méthodes ...  
}
```

Par convention <nom classe> commence par une majuscule.

La <protection> est facultative et sera étudiée plus tard. Pour l'instant nous utiliserons juste la protection **public** pour la classe où est définie le main.

Rq: une classe publique doit porter le même nom que le fichier source dans lequel elle est définie (il ne peut donc y avoir au plus qu'une classe publique dans un même fichier source).

## Définition d'attributs

Les attributs peuvent être vus comme des variables globales pour une classe. Ils sont visibles par toutes les méthodes de la classe.

La définition ressemble à celle de C (avec possibilité d'initialisation lors de la déclaration) :

```
<type> <nom variable> [ = <expr init> ] ;
```

Un *<type>* étant soit un *type primitif* soit un *type Classe*.

Ex: `int age;`

```
String nom;
```

```
float coeff = 1.2;
```

# Types primitifs

Type	bits	Plage de valeurs	description
<b>boolean</b>	1	<b>true ou false</b>	valeur logique
<b>char</b>	16	0 à 65535	Entier non signé sur 2 octets = Unicode
<b>byte</b>	8	-128 à +127	entier sur 1 octet
<b>short</b>	16	-32768 à +32767	entier sur 2 octets
<b>int</b>	32	- 2 147 483 648 à + 2 147 483 647	entier sur 4 octets
<b>long</b>	64	- 9 223 372 036 854 775 808 à + 9 223 372 036 854 775 807	entier sur 8 octets
<b>float</b>	32	$\pm 1.40239846e-45$ à $\pm 3.40282347e+38$	réel sur 4 octets (simple précision)
<b>double</b>	64	$\pm 4.94065645841246544e-324$ à $\pm 1.79769313486231570e+308$	réel sur 8 octets (double précision)

## Types primitifs

Les opérations sur les types primitifs sont plus performantes que sur les objets.

Les types: **byte**, **short**, **int** et **long** sont des entiers signés.

Le type **char** est aussi un type entier mais non signé. Il est initialement destiné à stocker un caractère étendu (16 bits). Comme en C, un caractère est encodé avec le code Unicode du caractère (sur-ensemble du code ASCII). On peut donc directement manipuler la valeur entière du caractère.

Ex: **char c = 'B' + 1;** idem que **char c = 66 + 1;**

Les types primitifs sont toujours passés par valeur aux méthodes. Ils ne sont donc pas modifiables par la méthode appelée.

# Déclaration de tableaux

Un tableau se déclare par

```
<type> [] <nom variable>;
```

Un tableau est en réalité un objet et doit donc s'initialiser grâce à l'opérateur **new** (que nous verrons plus tard) comme suit:

```
int [] t;                                ou plus simplement:  
t = new int [10];                         int [] t = new int [10];
```

On peut aussi initialiser un tableau: `int [] r = {5, 3, 6};`

Ou encore: `t = new int [] {x, 3, 12, y+3};`

Un tableau à n éléments est indicé de 0 à n-1. On peut connaître le nombre d'éléments d'un tableau par `<nom>.length`

Exemple: `int nbElem = t.length;`

Un tableau est en fait un objet et les objets sont toujours passés par référence. Ils sont donc *modifiables* par la méthode appelée.

# Définition des méthodes

Ici encore la syntaxe est proche du C:

```
<protection> <type> <nom méthode> ( <arg1>, ..., <argn>) {  
    ... corps de la méthode...  
}
```

Le **<type>** est le type renvoyée par la méthode ou **void** si elle ne renvoie rien.

Par convention le **<nom méthode>** commence par une minuscule.

**<argi>** sont les déclaration des paramètres formels. On utilise la même syntaxe que pour la déclaration d'attributs. Exemple:

```
int somme(int x, int y) {  
    return x + y;  
}
```

Le corps de la méthode est composé de déclaration de variables locales (même syntaxe que pour les attributs) et/ou d'instructions.

# Un premier programme

Fichier source: `HelloWorld.java`

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

```
C:\> javac HelloWorld.java  
  
C:\> dir  
09/09/2006  09:58  109 HelloWorld.java  
09/09/2006  10:03  425 HelloWorld.class  
  
C:\> java HelloWorld  
Hello World
```

# Lecture au clavier

Lecture au clavier: JDK définit une classe **Scanner** qui permet de lire les types primitifs et des chaînes depuis un flux d'entrée.

Créer un objet **Scanner** depuis **System.in**

```
Scanner sc = new Scanner(System.in);
```

Utiliser les méthodes suivantes:

```
public int    nextInt()
public float  nextFloat()
public double nextDouble()
public String nextLine()
```

Scanner peut faire beaucoup plus, ex: peut « découper » une entrée, peut lire depuis un **String**, ...  
(voir la documentation pour plus d'informations).

# Affichage à l'écran

Pour les affichages on utilise:

```
System.out.println("<message>" ou <expression>) ;
```

L'argument attendu de **println** est un **String**. Si ce n'est pas le cas il est automatiquement converti en **String** (si c'est un objet sa méthode **toString()** est appelée).

On peut donc utiliser l'opérateur + de concaténation de chaînes.

Exemple:

```
System.out.print("entrer un entier: ") ;
int x = sc.nextInt() ;
System.out.print("l'entier vaut: " + x) ;
```

NB: si on utilise **print** ou lieu de **println** le curseur reste après ce qui est affiché (il n'y a pas de passage à la ligne).

## Formatage de l'affichage avec printf

Java offre des facilités de formatage proche du `printf()` de C.

Comme en C: `printf(String format, args...)` où `format` est une chaîne à afficher pouvant inclure des parties fixes et des spécificateurs de format pour chaque argument `arg`. Chaque argument `arg` peut être un type primitif ou un objet.

NB : on verra plus tard que `String.format(format, args)` fournit une chaîne résultant du formatage (similaire à `sprintf()` de C).

En fait Java propose une classe `Formatter` qui crée un flux de sortie sur lequel on peut faire des `printf()` en utilisant la méthode `format` (c'est ce qui est utilisé par `String.format`).

On va voir étudier plus en détail comment formater un argument.

# Formatage de l'affichage avec printf

Chaque spécificateur commence par % et voici sa forme :

`%[flags][width][.precision]conversion`

Le champ *conversion* est le seul obligatoire, c'est un caractère qui indique comment doit être affiché l'argument **arg** :

**b**: booléen (**arg** peut être **boolean**, **Boolean**, quelconque).

**s**: chaîne (**arg** : quelconque - invoque **toString(arg)** ).

**c**: caractère (**arg** : **char**, **Character**, **byte**, **Byte**, **short**, **Short**).

**d** / **o** / **x** : entier en base 10 / base 8 / base 16 (**arg** : **byte**, **Byte**, **short**, **Short**, **int**, **Integer**, **long**, **Long**).

**f** / **e** / **g** : réel en notation flottante / scientifique / la plus appropriée (**arg**: **float**, **Float**, **double**, **Double**).

## Formatage de l'affichage avec printf

Le champ optionnel *width* indique la longueur minimum de sortie. Si le résultat est moins long il est justifié à droite et complétée (à gauche) par des espaces. On peut utiliser le *flag* + pour justifier à gauche (et donc remplir à droite) et le *flag* 0 pour compléter la sortie par des zéros (pour les nombres).

Le flag + émet le signe + pour les nombres positifs. Le flag ' ' (espace) émet un espace devant les nombres positifs.

Le champ optionnel *precision* indique le nombre de décimales pour les réels. Pour **s** elle indique une longueur maximum d'affichage (au-delà la sortie est tronquée).

# Formatage de l'affichage avec printf

System.out.printf("<%d>" ,256) ;	<256>
System.out.printf("<%5d>" ,256) ;	< 256>
System.out.printf("<%05d>" ,256) ;	<00256>
System.out.printf("<%-5d>" ,256) ;	<256 >
System.out.printf("<%+5d>" ,256) ;	< +256>
System.out.printf("<%5x>" ,26) ;	< 1a>
System.out.printf("<%5X>" ,26) ;	< 1A>
System.out.printf("<%f>" ,Math.PI) ;	<3,141593>
System.out.printf("<%7.3f>" ,Math.PI) ;	< 3,142>
System.out.printf("<%s>" , "bonjour") ;	<bonjour>
System.out.printf("<%10s>" , "bonjour") ;	< bonjour>
System.out.printf("<%-10s>" , "bonjour") ;	<bonjour >
System.out.printf("<%4.4s>" , "bonjour") ;	<bonj>

# Instruction d'affectation et expressions

On utilise comme en C :

<variable> = <expr>                      <variable> <op>= <expr>

Une expression numérique est constituée de constantes numériques, de variables entières ou réelles et des opérateurs :

+ - \* / % ++ --

Une expression sur les chaînes peut utiliser l'opérateur + pour concaténer 2 chaînes.

Une expression booléenne est constituée de constantes (**false**, **true**), de variables booléennes et des opérateurs :

== != < <= > >= (comparaisons)  
&& (ET) || (OU) ! (NON)

Les expressions booléennes apparaissent aussi comme tests dans l'instruction conditionnelle (**if**) et les boucles (**while**, **for**, ...).

# Instruction conditionnelle

Tout comme en C:

```
if (<expr booléenne>)
    <instruction si vrai>
```

```
if (<expr booléenne>)
    <instruction si vrai>
else
    <instruction si faux>
```

Rappel: si la partie ALORS (resp. SINON) nécessite plus d'une instruction on utilisera un bloc { } .

Exemple:

```
if (a > b && a > c) {
    m = a;
    u = b + c;
} else
    u = a + b + c;
```

# Instructions itératives

Comme en C on dispose de 2 boucles principales qui répètent un traitement **tant que** la condition est vraie.

**while** (<test>  
    <instruction à itérer>

On peut ne jamais passer  
dans la boucle

Le <test> est évidemment une expression booléenne.

Rappel: on utilisera un bloc { } s'il y a plusieurs instructions à itérer (i.e.  
dans le corps de la boucle).

**do**  
    <instruction à itérer>  
**while** (<test>)

On passe au moins 1 fois  
dans la boucle

## Instruction itérative

La boucle **for** est prévue pour gérer un (ou plusieurs) compteur(s).

```
for (<expr init> ; <test> ; <expr fin iter>)
    <instruction à itérer>
```

Elle est équivalente à la boucle **while** suivante:

```
<expr init> ;
while (<test>) {
    <instruction à itérer>;
    <expr fin iter>;
}
```

On exécute d'abord **<expr init>** (peut être vide). Tant que **<test>** est vrai (si vide c'est true) on exécute l'**<instruction à itérer>** suivie de l'**<expr fin iter>** (peut être vide). Et on recommence.

## Exemple de boucle for

Saisie d'un tableau d'entiers:

```
for (int i = 0; i < t.length; i++) {  
    System.out.print("entrer i[" + i + "]: ");  
    t[i] = c.readInt();  
}
```

Boucle infinie:

```
for (;;) {  
    ...  
}
```

## Instructions de rupture de boucle

L'instruction **break** sort de la boucle dans laquelle elle se trouve. Cela revient donc à sauter à l'instruction suivant la boucle.

L'instruction **continue** permet de passer à l'itération suivante. Dans le cas des boucles **while** et **do...while** cela revient à sauter au <test> de boucle. Pour la boucle **for**, on exécute <expr fin iter> et ensuite on saute au <test>.

Si on veut que **break** ou **continue** s'applique à une boucle de plus haut niveau (dans le cas de boucles imbriquées) on place un label avant la boucle voulue (pour la nommer) et on utilise

```
break <label>
continue <label>
```

## Exemple de break avec label

```
int [][] t = new int [5][10];
boolean trouve = false;
...
sortie: // on place une label
    for (int i = 0; i < t.length; i++) {
        for (int j = 0; j < t[0].length; j++) {
            if (t[i][j] == x) {
                trouve = true;
                break sortie;
            }
        } // fin de boucle interne
    } // fin de boucle externe
```

# Instruction de sélection

Cas général:

```
switch (<expr>) {  
    case <expr cst>:  
        <instructions>  
        break;  
        ...  
    case <expr cst>:  
        <instructions>  
        break;  
    default:  
        <instructions>  
}
```

Att: en l'absence de **break**, après l'exécution des instructions d'un cas, on exécute les instructions du cas suivant (jusqu'à trouver un **break**).

# **PROGRAMMATION**

## **ORIENTEE**

### **OBJET (BASES)**

# Introduction

Programmation OO: assembler des petits éléments (objets) autonomes pour en construire de plus grands.

Ecrire un programme en OO: définir des objets (classes) et les utiliser ou réutiliser des objets existants.

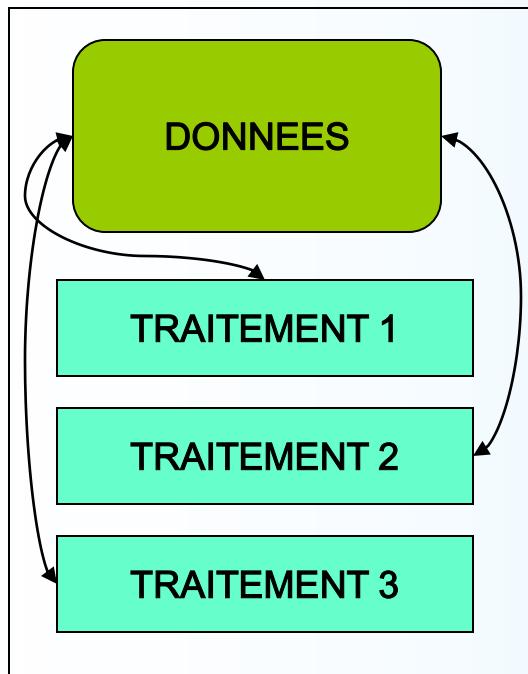
L'utilisation d'un objet se fait de manière organisée via les méthodes (et/ou les attributs) prévues par l'objet.

On ne sépare plus les données des traitements portant sur ces données.  
Objet = Données + Programmes.

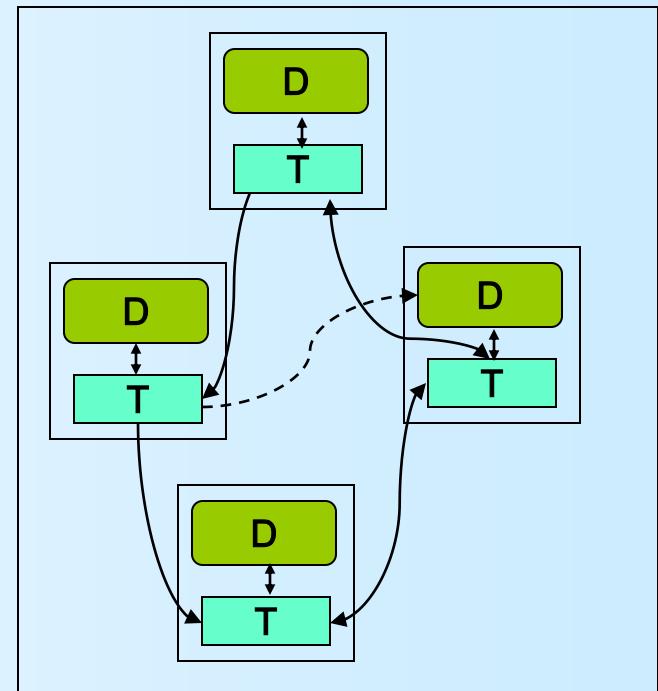
*Encapsulation:* on "cache" des données à l'intérieur d'une classe en ne permettant de les manipuler qu'au travers des méthodes de la classe.

# Programmation impérative vs programmation OO

Programme impératif: ensemble de fonctions effectuant des opérations sur des données communes: données et traitements sont séparés



Programme OO: ensemble d'objets contenant des données ET des traitements sur ces données. Ces objets s'utilisent entre eux.



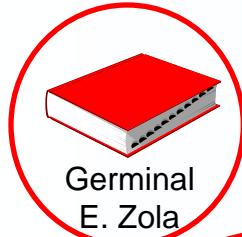
# Objet

Approche impérative : « que doit faire mon programme ? »

Approche OO : « de quoi doit être composé mon programme ? »

*Cette composition est conséquence d'un choix de modélisation fait pendant la conception*

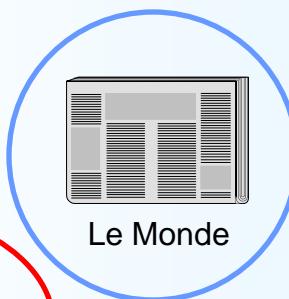
Exemple: objets d'une bibliothèque



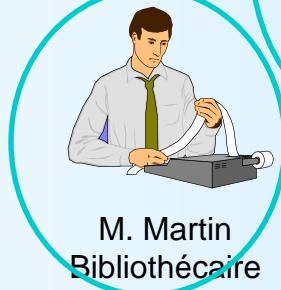
Germinal  
E. Zola



Le seigneur des anneaux  
J.R.R. Tolkien



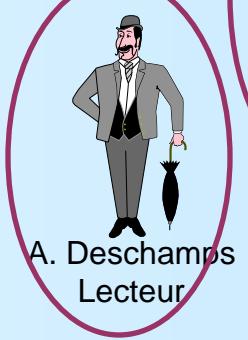
Le Monde



M. Martin  
Bibliothécaire



A. Dupont  
Directrice



A. Deschamps  
Lecteur

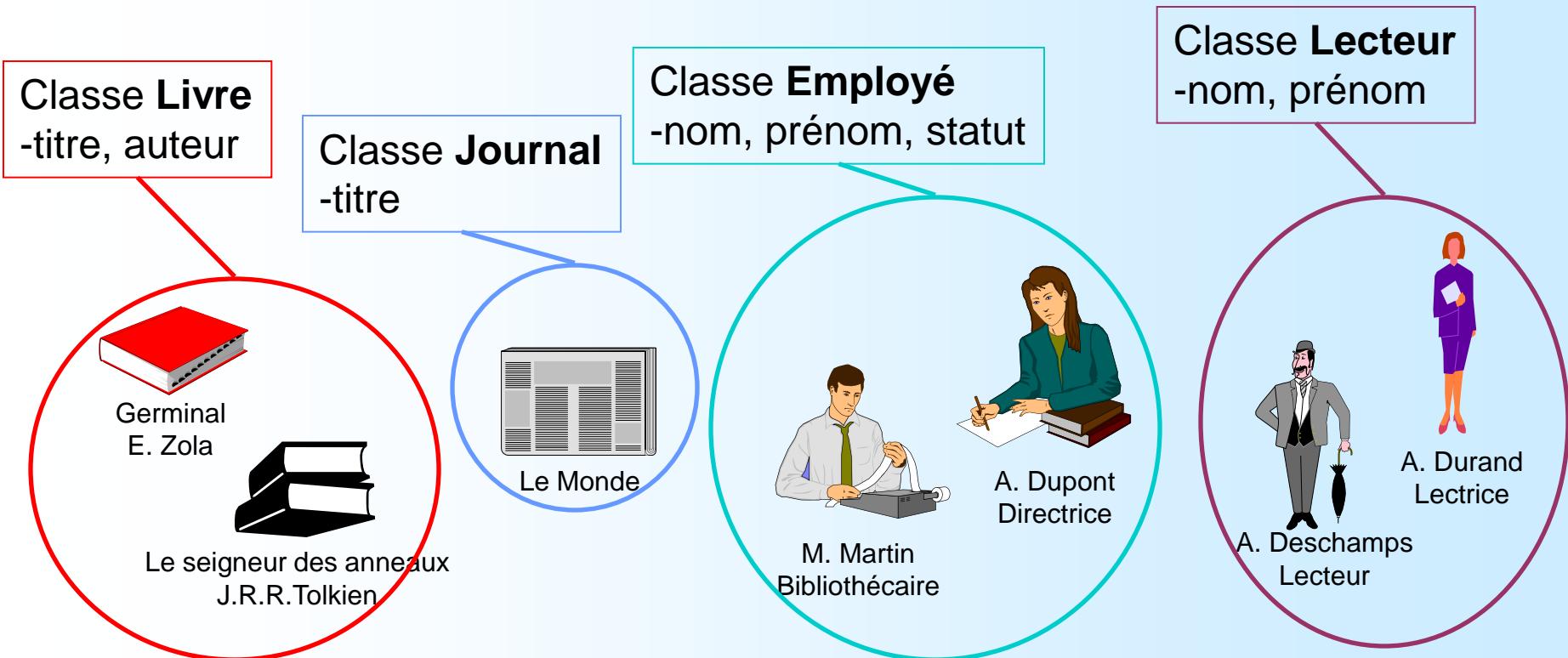


A. Durand  
Lectrice

# Classe

Des objets similaires peuvent être décrits par une même *abstraction* : une *classe*

- même structure de données et méthodes de traitement
- valeurs différentes pour chaque objet



# Classe

Une *classe* est un modèle commun à un ensemble d'objets de mêmes caractéristiques.

Une *instance* (de classe) est l'autre nom d'un objet réel. C'est une manifestation concrète d'une classe.

Une classe est une représentation abstraite d'un objet et une instance en est une représentation concrète.

Ex: on a une classe Employé (avec un nom, prénom, statut) et 2 objets (instances): A. Dupont, directrice et M. Martin, bibliothécaire.

## Contenu d'une classe

Une classe définit donc les caractéristiques communes à un ensemble d'objets similaires. Elle se compose de 2 parties :

- les attributs (données): caractéristiques individuelles qui différencient un objet d'un autre. On les appelle aussi *variables d'instance*.  
Ex: un Lecteur a pour attribut: un **nom**, un **prénom**
- les méthodes (programmes): ce sont les traitements auxquels un objet peut répondre.  
Ex: un Lecteur peut **emprunter** ou **retourner** un ouvrage

# Protection

Le but de la protection est de limiter la visibilité (et donc l'accessibilité) d'un attribut ou d'une méthode. Java propose 4 niveaux de protection (de la moins restrictive à la plus restrictive) :

- **public**: est visible partout.
- par défaut: n'est visible que dans les classes du même *package*.
- **protected**: n'est visible que depuis les sous-classes.
- **private**: n'est pas visible hors de la classe.

# Définition de classe

Comme on l'a vu:

```
[<protection>] class <nom classe> {  
    ... définition d'attributs ...  
    ... définition de méthodes ...  
}
```

Exemple:

```
public class Lecteur {  
    public String nom;  
    private String prenom;  
    void emprunte(Ouvrage o) {  
    }  
    void retourne(Ouvrage o) {  
    }  
}
```

*type et nom* définis dans la classe.  
*valeur* définie dans l'instance.

## Création d'une instance

Pour obtenir un nouvel objet d'une classe on utilise l'opérateur  
**new <Classe> ()**

Ex: **Lecteur l = new Lecteur();**

A partir d'un objet on peut accéder aux attributs et/ou méthodes grâce à une notation pointée: **<var objet>.<membre>**:

Ex: **l.nom = "Deschamps";**      Rq: on évite !

**l.emprunte(o);**

A l'intérieur d'une même classe on peut accéder directement aux attributs et méthodes (pas besoin de notation pointée) mais il est parfois nécessaire de faire référence à l'objet courant. On utilise alors le mot clé **this** qui désigne l'objet courant.

# Constructeur

Quel est l'effet de **new** ? Allocation de dans le tas (heap) de l'espace nécessaire pour le nouvel objet. Exécution du constructeur approprié pour initialiser l'objet (par exemple ses attributs).

Par défaut toute classe possède un constructeur (sans arguments) qui initialise toutes les variables d'instances (attributs) à des valeurs par défaut suivant leurs types:

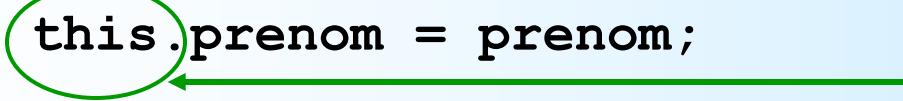
nombres: 0      booléen: **false**      objets: **null**

On peut définir ses propres constructeurs, ils se définissent comme des méthodes de même nom que la classe mais **sans type de retour**.

## Définition d'un constructeur

Exemple: définition d'un constructeur dans la classe Lecteur:

```
public class Lecteur {  
    private String nom;  
    private String prenom;  
  
    public Lecteur(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
}
```



utilisation de **this**  
pour atteindre l'attribut

Utilisation:

```
Lecteur l = new Lecteur("Deschamps", "Antoine");
```

## Définition de plusieurs constructeurs

Tout comme pour les méthodes il est possible de définir plusieurs constructeurs avec des arguments de type différent. On appelle cela la *surcharge*.

```
public class Lecteur {  
    private String nom;  
    private String prenom;  
  
    public Lecteur(String nom, String prenom) {  
        ...  
    }  
    public Lecteur(String nom) {  
        this(nom, null);          on fait appel à l'autre  
    }                                constructeur.  
}                                    doit être la 1ère instruction
```

## Méthodes d'accès

On évite de déclarer les attributs en **public** (encapsulation). On définit alors des *méthodes d'accès*, appelées *accesseurs (getter)* ou *mutateurs (setter)* pour les attributs qui le nécessitent.

```
public class Lecteur {  
    private String nom;  
    private String prenom;  
    ...  
    public String getNom() {  
        return nom;  
    }  
    public void setNom(String nom) {  
        this.nom = nom;  
    }  
}
```

# Objets et références

Lors de la création d'un objet avec new on récupère une référence (c-à-d un pointeur) sur une zone mémoire contenant les membres (variables d'instances et méthodes) de l'objet. Ainsi tous les objets sont manipulés implicitement via des pointeurs. Attention: les opérateurs d'affectation = et de comparaisons == et != ne travaillent que sur les références, pas les contenus.

```
Lecteur x = new Lecteur("Deschamps", "Antoine");
Lecteur y = new Lecteur("Deschamps", "Antoine");

if (x == y)           vaut faux !
...
```

Il faut définir un méthode (ou mieux redéfinir `equals`) qui permet vraiment de comparer 2 objets.

# Documentation en Java

## Documentation dans le code

Problème : si la documentation et le code sont gérés séparément on a tendance à ne pas mettre à jour la documentation au fur et à mesure que le code évolue (la documentation devient obsolète)

Solution : ne pas séparer la documentation du code.

Adapté à la documentation d'API (Application Programming Interface).

Mise en œuvre :

1. Rédiger la doc sous forme de commentaires dans les fichiers sources (ces commentaires sont donc ignorés par le compilateur) en utilisant des balises ou des tags spécifiques (commentaires structurés)
2. Utiliser un outil qui va analyser les commentaires, en extraire la documentation et produire un fichier de sortie (par exemple du HTML)

Ex: Javadoc pour Java, Doxygen pour C++, pldoc (Prolog), Edoc (Erlang)...

## L'outil Javadoc

Javadoc est fourni dans le JDK,

<http://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>

Génère la documentation sous la forme de fichiers HTML

Les commentaires Javadoc sont de la forme `/** ...commentaire... */`

Un commentaire Javadoc avant la définition de la classe décrit la classe

Un commentaire Javadoc avant la définition d'un constructeur le décrit

Un commentaire Javadoc avant la définition d'une méthode la décrit

Pour décrire un package on crée un fichier `package-info.java` qui ne contient qu'un commentaire.

Javadoc reconnaît des *tags* Javadoc de la forme `@tag` qu'il traite et formate dans le fichier de sortie (en HTML).

## Tags Javadoc

Principaux *tags* reconnus par javadoc pour la description :

<b>@author</b>	Nom de l'auteur
<b>@param</b>	Définit un paramètre de méthode
<b>@return</b>	Définit la valeur de retour (si pas void)
<b>@throws</b>	Indique que la méthode lève une exception
<b>@exception</b>	Indique que la méthode laisse passer une exception
<b>@deprecated</b>	Indique que la méthode est dépréciée (ne plus l'utiliser)
<b>@see</b>	Renvoie à la doc d'une autre méthode ou classe
<b>@since</b>	Indique depuis qu'elle version du JDK c'est disponible

## Javadoc et HTML

Attention: l'intérieur d'un commentaire `/** ... */` doit être écrit en HTML.

Ceci permet une mise en page soignée grâce à la puissance des balises HTML de la forme `<balise>`.

Attention aux paragraphes (à marquer explicitement avec `<p>`).

Attention aux caractères spéciaux HTML, par exemple pour produire `<` et `>`.

Solutions:

- Comme en HTML, ex: `List<Personne>;`
- Utiliser le tag `@literal`, ex `{@literal List<Personne>}`
- Utiliser le tag `@code`, ex `{@code List<Personne>}` qui générera en plus une fonte appropriée pour le code et permet des sauts de ligne.

## Exemple

```
/**  
 * Une classe pour définir un {@code Lecteur}  
 * @author Daniel Diaz  
 */  
  
public class Lecteur {  
    /**  
     * Emprunte un ouvrage  
     * @param o l'ouvrage à emprunter  
     * @return indique si l'opération s'est bien passée  
     */  
    public boolean emprunte(Ouvrage o) {  
        ...  
    }  
}
```

Exemple : cf la documentation de la classe **String** (dispo dans l'EPI)

# VARIABLES / METHODES

DE

# CLASSE

## Variables de classe

Les attributs d'une classe donnent lieu à des variables stockant les informations d'un objet. On les qualifie de **variables d'instance** car elles n'existent qu'une fois que l'on a instancié la classe (pour obtenir un objet).

On a parfois besoin de variables qui soient communes à toutes les instances d'une classe. On parle alors de **variable de classe**. Il faut voir une variable de classe comme une variable globale visible par tous les objets issus de cette classe.

On déclare une telle variable avec le mot clé **static**.

Exemple: on veut définir une classe **Ouvrage** de telle sorte que tout ouvrage créé se voit attribué un numéro séquentiel (1, 2, ...).

## Variables de classe: exemple

```
public class Ouvrage {  
    private static int noOuvrage = 0;  
  
    protected int numero;  
    protected String titre;  
  
    public Ouvrage(String titre) {  
        numero = ++noOuvrage; un numéro séquentiel  
        this.titre = titre;  
    }  
}
```

## Variables de classe: initialisation

Une variable de classe est allouée et initialisée au début du programme (lorsque la classe est chargée en mémoire). On peut initialiser une variable de classe lors de sa déclaration:

```
private static int noOuvrage = 0;
```

Lorsque on a une initialisation complexe on peut déclarer un bloc d'instruction pour réaliser cette initialisation. On déclare un tel bloc avec **static { ... }**. Il est exécuté au chargement de la classe.

```
private static int tbl[] = new int [10];
static {
    for(int i = 0; i < tbl.length; i++)
        tbl[i] = 2 * i;
}
```

## Méthodes de classe

Similairement on peut définir des **méthodes de classe** (par opposition aux méthodes d'instances). Ces méthodes sont donc accessibles sans avoir besoin d'instancier la classe.

Elles aussi se définissent avec le mot clé **static** (ex: **main**).

Exemple: on peut connaître le nombre d'ouvrages en consultant la variable **noOuvrage**. On donne donc une méthode de classe:

```
public class Ouvrage {  
    private static int noOuvrage = 0;  
  
    public static int getNbOuvrage() {  
        return noOuvrage;  
    }  
    ...  
}
```

## Méthodes de classes

Les variables/méthodes de classe sont directement visibles depuis une instance de la classe.

Si elles sont déclarées **public**, elle peuvent aussi être utilisées depuis n'importe avec la notation pointée: <Classe>.<membre>.

```
int nbOuvrage = Ouvrage.getNbOuvrage();
```

La classe **Math** fournit plusieurs fonctions mathématiques. Ce sont de méthodes de classe.

```
double x = Math.sqrt(y);
```

```
int diff = Math.abs(a-b);
```

# LES CHAINES DE CARACTERES

## La classe String

La manipulation des chaînes se fait via la classe **String**. On peut créer une chaîne en appelant explicitement `new String(...)` ou implicitement en utilisant une constante chaîne (ex: "bonjour").

Remarque: la classe **String** permet de stocker une chaîne et d'extraire (retourner) plusieurs types d'informations (longueur, recherche d'un caractère dans la chaîne,...) mais elle ne permet pas de modifier la chaîne. Pour cela il faut utiliser **StringBuffer**.

Java offre l'opérateur `+` sur les chaînes de caractères pour faire la concaténation. Toutes les autres opérations se font via les méthodes de la classe **String**. Nous en étudions ici quelques une (voir la documentation pour plus d'informations).

## Constructeurs de la classe String

**String()** : crée une chaîne vide (idem que "")

**String(String original)** : crée une copie de la chaîne **original**.

**String(char[] value)** : crée une chaîne à partir d'un tableau de caractères.

**String(char[] value, int offset, int count)** : crée une chaîne à partir d'un sous-tableau de caractères (en prenant les **count** éléments à partir de l'indice **offset**).

**String(StringBuffer buffer)** : crée une chaîne à partir d'un **StringBuffer**.

# Méthodes de la classe String

**int length()** : retourne la longueur de la chaîne

**char charAt(int index)** : retourne le caractère à la position **index** (à partir de 0)

**boolean equals(Object obj)** : teste l'égalité entre la chaîne courante et un autre objet (supposé être un **String**).

**boolean equalsIgnoreCase(String str)** : teste l'égalité sans différencier minuscules et majuscules

**int compareTo(String str)**

**int compareToIgnoreCase(String str)** : compare la chaîne courante et **str**. Retourne 0 si égales, < 0 inférieure, > 0 si supérieure.

**boolean startsWith(String str)**

**boolean endsWith(String str)** : teste si la chaîne commence / termine par **str**.

# Méthodes de la classe String

**int indexOf(int ch) / int lastIndexOf(int ch) /  
int indexOf(String str) / int lastIndexOf(String str)** : retourne la position de la première/dernière occurrence du caractère **ch** / chaîne **str**.

**String substring(int pos) / String substring(int pos, int endpos)** : retourne la sous-chaîne commençant à **pos** jusqu'à la fin (ou jusqu'à **endPos-1**).

**String replace(char ch1, char ch2)** : retourne la chaîne obtenue en remplaçant **ch1** par **ch2**.

**String toLowerCase() / String toUpperCase()** : retourne la chaîne convertie en minuscule/majuscule.

## Méthodes de la classe String

**String trim()** : retourne la chaîne sans les espaces du début et de fin.

**char[] toCharArray()** : retourne les caractères de la chaîne.

**static String format(String format, Object... args)** : cette méthode de classe permet de créer une chaîne résultante à partir de format et args à la manière de la fonction **sprintf** de C.

**static String valueOf(...)** : cette méthode de classe permet de convertir tous les types primaires en chaîne. Exemple:

```
int i = 100;  
String stri = String.valueOf(i);
```

AUTRES  
CLASSES  
UTILES

# La classe Math

Dans ce qui suit *T* désigne un type `int`, `long`, `float`, `double`. Les fonctions suivantes sont des méthodes de classes (`static`).

**Pi**: variable de classe de type `double` valant pi (3,14159265...).

**E**: variable de classe de type `double` valant e (2,718281828...).

`double sin() / cos() / tan() / asin() / acos() / atan()` : fonctions trigonométriques (angles en radians).

`T max(T n1, T n2) / min(T n1, T n2)` : retourne le min / max entre n1 et n2.

`T abs(T x)` : retourne la valeur absolue de x.

`double sqrt(double x) / cbrt(double x)` : retourne la racine carrée / cubique de x.

# La classe Math

**double exp(double x)** : retourne  $e^x$ .

**double log(double x) / log10(double x)** : retourne le log en base  $e$  / base 10 de **x**.

**double pow(double x, double n)** : retourne  $x^n$ .

**double IEEEremainder(double x, double y)** : retourne le reste de la division de **x** par **y**.

**double ceil(double x) / double floor(double x)** : retourne l'arrondi par excès / défaut de **x**.

**double random()** : retourne un nombre aléatoire en  $[0,1[$ . Voir aussi la classe **Random**.

## Les classes « enveloppeurs »

Les classes « enveloppeurs » (*wrappers*) sont prévues pour pouvoir considérer un type primitif comme un objet. Elles sont constitués d'une seule variable d'instance pour enregistrer l'information. Elles fournissent :

- un constructeur à partir du type primitif et un constructeur à partir d'une chaîne (**String**).
- une méthode pour récupérer la valeur primitive dans son type ou un autre (conversion) **typeValue** (ex: **intValue()**).
- la méthode **equals** / **compareTo** pour la comparaison.
- une méthodes de classe pour convertir une chaîne en un type primitif **parseType** (ex: **int Integer.parseInt(String s)**).

## Exemple d'utilisation de la classe Integer

```
int age = ...;  
...  
Integer obj1 = new Integer(age);  
...  
p.empiler(obj1);  
...  
Integer obj2 = (Integer) p.depiler();  
...  
age = obj2.intValue();  
...
```

## Les flux standards

La classe **System** définit 3 variables de classe qui permettent d'utiliser les flux d'entrée/sortie standards du système d'exploitation.

**in**: de type **InputStream** elle correspond à l'entrée standard (par défaut: clavier).

**out**: de type **PrintStream** elle correspond à la sortie standard (par défaut c'est la console courante).

**err**: de type **PrintStream** elle correspond à la sortie d'erreurs (par défaut c'est la console courante).

Ces 3 flux peuvent être redirigés via **setFlux** (ex: **setOut(...)** ).

On a déjà utilisé **print** / **println**, depuis le JDK 1.5 il existe aussi **printf**.

# HERITAGE

# Héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :

- la classe mère (ou super-classe)
- la classe fille (ou sous-classe) qui hérite d'attributs / méthodes de la classe mère.

En Java, une classe (fille) ne peut hériter que d'une seule classe mère (héritage simple). Une classe mère peut par contre avoir plusieurs classes filles.

On obtient ainsi une hiérarchie (ou arborescence) de classes. Par défaut toute classe hérite de la classe la plus générale: **Object**.

# Héritage

Le concept d'héritage permet donc:

La réutilisation: une sous-classe possède les attributs / méthodes de la classe mère (sauf ceux déclarés **private**).

La spécialisation: on peut aussi redéfinir certains attributs / méthodes de la classe mère dans la sous-classe.

L'extension: on peut ajouter de nouveaux attributs / méthodes propres à la sous-classe.

# Héritage

La déclaration en Java se fait comme suit:

```
[<protection>] class <classe> extends <super classe> {  
    ... (re)définition d'attributs ...  
    ... (re)définition de méthodes ...  
}
```

Remarques :

On n'hérite pas des constructeurs de la classe mère ! Si on en a besoin il faut redéfinir ce constructeur dans la classe fille.

Par défaut le constructeur implicite (sans argument) de la classe mère est appelé (il doit donc exister). On peut toutefois appeler un constructeur de la classe mère en utilisant : **super (<arguments>)** . Le mot-clé **super** permettant de référencer les membres de la classe mère.

## Héritage (exemple)

```
public class Ouvrage {  
    private static int noOuvrage = 0;  
  
    protected int numero;  
    protected String titre;  
  
    public Ouvrage(String titre) {  
        numero = ++noOuvrage; un numéro séquentiel  
        this.titre = titre;  
    }  
}
```

## Héritage (exemple)

```
public class Livre extends Ouvrage {  
    private String auteur;  
  
    public Livre(String titre, String auteur) {  
        super(titre);           doit être la 1ère instruction  
        this.auteur = auteur;  
    }  
  
    public Livre(String titre) {  
        this(titre, null);  
    }  
}
```

# Compilation ≠ Exécution

L'héritage offrant la possibilité de redéfinir des membres dans une sous-classe il faut comprendre comment fonctionne Java à 2 moments différents :

- à la compilation : Java vérifie la cohérence au niveau des types (classes). On parle de **typage statique**. Le but est de détecter des anomalies de programmation. Le typage statique est utilisé pour *résoudre les variables*.
- à l'exécution : Java utilise un mécanisme de **liaison dynamique** pour trouver la méthode pertinente. La liaison dynamique est utilisée pour *résoudre les appels de méthodes*.

Il est clair que à la compilation (statiquement) on peut déduire moins de chose que à l'exécution (dynamiquement), donc :

- Il faut alors parfois expliciter au compilateur ce qu'on veut faire grâce à l'opérateur de transtypage (**cast**).
- il se peut que des erreurs soient découvertes à l'exécution.

# Typage statique : l'opérateur cast

Supposons **x** déclaré comme un objet de classe **C**.

```
C x;
```

On peut évidemment lui affecter n'importe quel objet de même type **C** (ex: par **new C()**). Mais pour exploiter l'héritage on peut aussi vouloir lui affecter un objet **y** de type **D** qui est :

- plus spécialisé (c-à-d **D** est une sous-classe de **C**). La conversion de type est alors implicite, on peut simplement écrire: **x = y;**
- plus général (c-à-d **D** est une super-classe de **C**). La conversion doit alors être expliciter avec l'opérateur de cast. **x = (C) y;**

En conclusion on peut affecter un objet à un objet « plus général ». Pour faire l'inverse on doit utiliser un cast.

Rq: On ne peut pas faire un cast s'il n'y a pas de lien d'héritage entre **C** et **D**.

## Utilisation de cast: erreur de compilation

```
class A {           class B extends A {  
    int x = 1;       int y = 2;  
}  
  
A a = new B();    cast inutile: B est une sous-classe de A  
...  
int z = a.y;      erreur à la compilation
```

Le compilateur émet une erreur (`cannot find variable y`) car la variable `y` n'est pas définie dans `A` (ni dans une de ses super-classes). Bien que nous sachions que `a` est en fait du type `B`, le compilateur ne le devine pas. On utilise le cast:

```
int z = ((B) a).y;
```

ATT: l'opérateur `.` est plus prioritaire qu'l'opérateur (cast) il faut donc utiliser des parenthèses pour forcer l'ordre d'évaluation.

## Utilisation de cast: erreur à l'exécution

```
class A {                                class B extends A {  
    int x = 1;                          int y = 2;  
}  
  
A a = new A();  
...  
int z = ((B) a).y;                      erreur à l'exécution
```

Déclenche une exception (`java.lang.ClassCastException: A`) pour signaler l'erreur. En effet, `a` est de type `A` et ne peut être converti en `B`.

Remarque: cette erreur n'est pas détecté à la compilation.

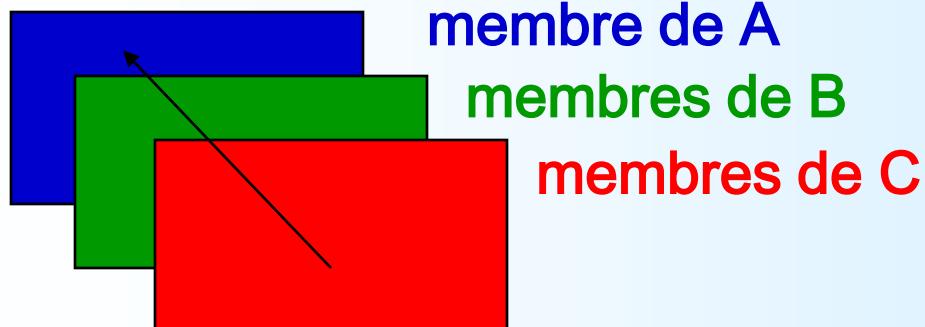
# Résolution des variables

La résolution des variables est faite par typage statique. On détermine donc au moment de la compilation la variable utilisée.

Supposons une classe A super-classe de B super-classe de C. Si on a un objet typé comme C alors on cherchera la définition de la variable dans C, puis dans B, puis dans A.

Si le même objet est typé B on cherchera dans B puis dans A.

Si le même objet est typé A on cherchera dans A.



```

class A {
    int x = 1;
}

class C extends B {
    ...
}

C u = new C();
System.out.println("x = " + u.x);

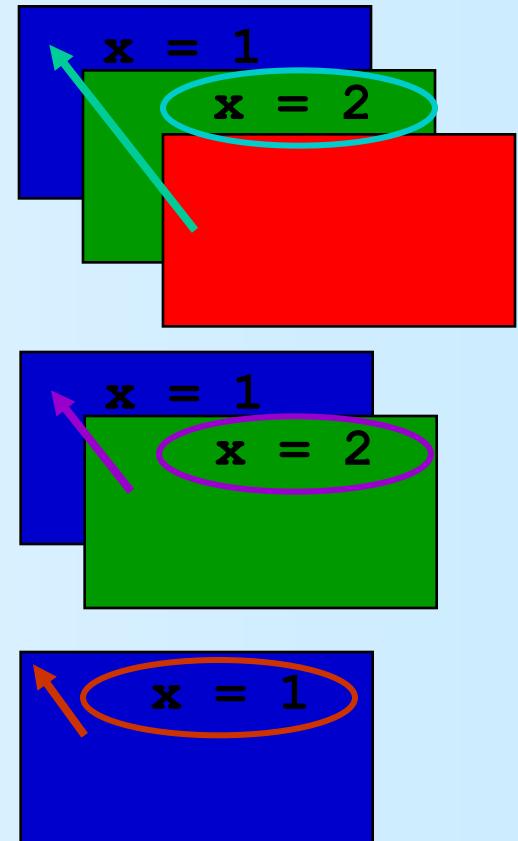
B v = u;
System.out.println("x = " + v.x);

A w = u;
System.out.println("x = " + w.x);

```

Affichera:

2  
2  
1



## Résolution des méthodes

La résolution des appels de méthodes est faite par le mécanisme de liaison dynamique. On détermine au moment de l'exécution la méthode à appeler (c'est donc différent du traitement des variables).

L'idée de la liaison dynamique (ou liaison retardée – *late binding*) est de déterminer quel est le vrai type d'un objet pour déterminer la méthode à appeler. Ceci ne peut se faire que dynamiquement à l'exécution.

Ce mécanisme se justifie par le fait qu'on veut généralement qu'un objet conserve ses comportements même si on le stocke dans un objet plus général (et donc que ce soit ses méthodes propres qui soient invoquées).  
**Ca marche donc comme on s'y attend !**

```

class A {
    void foo() {
        System.out.print("foo A");
    }
}
class C extends B {
    ...
}

```

**C u = new C();  
u.foo();**

**B v = u;  
v.foo();**

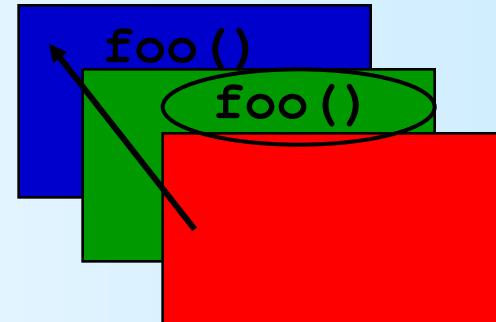
**A w = u;  
w.foo();**

```

class B extends A {
    void foo() {
        System.out.print("foo B");
    }
}

```

dans les 3 cas le type réel de l'objet à l'exécution est C



Affichera:

**foo B  
foo B  
foo B**

# La classe Object

Toute classe hérite (directement ou indirectement) de la classe **Object**. Voici quelques méthodes qu'offre cette classe :

**String toString()** : retourne une chaîne associé à l'objet. Cette méthode peut être appelée explicitement. Elle est invoquée implicitement à chaque fois que l'objet est passé et qu'une chaîne est attendue. Par exemple dans **System.out.println**. Peut bien sûr être redéfinie.

**boolean equals(Object obj)** : retourne vrai si que l'objet est le même que **obj**. Cette méthode teste **this == obj** (même adresse) et doit donc généralement être redéfinie.

**void finalize()** : est exécuté juste avant la libération de l'objet par le GC. Peut-être redéfinie.

**Class getClass()** : retourne la classe réelle de l'objet (introspection). Peut être utilisé dans une redéfinition de **equals**.

## Redéfinition de `toString`

```
public class Lecteur {  
    protected String nom;  
    protected String prenom;  
    ...  
    public String toString() {  
        return "lecteur: nom = " + nom +  
               ", prénom = " + prenom;  
    }  
}  
...  
Lecteur x = new Lecteur("Deschamps", "Antoine");  
System.out.println(x);
```

Affichera :

`lecteur: nom = Deschamps, prénom = Antoine`

# Redéfinition de equals

```
public class Lecteur {  
    ...  
    public boolean equals(Object obj) {  
        if (obj == null || !(obj instanceof Lecteur))  
            return false;  
        Lecteur l = (Lecteur) obj;  
        return nom.equals(l.nom) &&  
              prenom.equals(l.prenom);  
    }  
}  
  
Lecteur x = new Lecteur("Deschamps", "Antoine");  
Lecteur y = new Lecteur("Deschamps", "Antoine");  
if (x == y)          on a vu que vaut faux !  
if (x.equals(y))    mais equals vaut vrai !
```

Remarque : on peut tester `getClass() != obj.getClass()` au lieu de `!(obj instanceof Lecteur)` pour ne pas capturer une sous-classe

# CLASSES ABSTRAITES METHODES ABSTRAITES ET INTERFACES

# Classes et méthodes abstraites

On a vu qu'une classe contient la définition de variables (attributs) et de méthodes.

On peut vouloir déclarer une méthode sans en donner son implémentation. Ceci dans le but de forcer toute classe héritière à implémenter cette méthode. On parle alors de méthode abstraite et par extension de classe abstraite.

Une **méthode abstraite** est donc une méthode pour laquelle on ne définit pas son implémentation (le corps de la méthode est absent).

Une classe comportant au moins une méthode abstraite est dite **classe abstraite**.

## Classes et méthodes abstraites

On utilise le même mot-clé **abstract** pour déclarer une classe abstraite et une méthode abstraite. Exemple :

```
abstract public class Polygone {  
    ...  
    public double perimetre() {  
        ...  
        return ...;  
    }  
  
    abstract public double surface();  
}
```

Remarque: une classe abstraite *ne peut pas être instanciée*:

**Polygone p = new Polygone(); Erreur de compilation**

## Classes et méthodes abstraites

La seule utilisation d'une classe abstraite est donc l'héritage. Lorsqu'une classe hérite d'une classe abstraite elle doit fournir une implémentation pour toutes les méthodes abstraites de la classe abstraite (les autres méthodes peuvent si besoin être redéfinies mais cela n'est évidemment pas obligatoire). Exemple :

```
public class Rectangle extends Polygone {  
    protected Point sg, sd, id, ig;  
    ...  
    public double surface() {  
        double l1 = sg.distance(sd);  
        double l2 = sg.distance(ig);  
        return l1 * l2;  
    }  
}
```

## Classes abstraites : utilité

Les intérêts d'une classe abstraite sont :

- de limiter son usage à l'héritage (ne peut-être instanciée).
- pour faire de la factorisation de code (ce qui est implémenté) tout en ne donnant qu'une implémentation partielle.
- pour ce qui n'est pas implémenté (méthodes abstraites) on spécifie toutefois la signature de la méthode. C'est-à-dire ce qu'il faut savoir pour l'utiliser (ce qu'on doit fournir à la méthode via les arguments et ce que l'on récupère via la valeur de retour).
- d'obliger les classes qui héritent à fournir une implémentation pour les méthodes abstraites.

# Interfaces

Une **interface** est assimilable à une classe abstraite ne contenant que des déclarations de méthodes abstraites. Il est alors inutile de spécifier les **abstract**. On définit une interface avec le mot clé **interface**. Exemple :

```
public interface Polygone {  
    ...  
    public double perimetre();  
    public double surface();  
}
```

Une interface ne peut pas contenir de déclarations de variables mais on peut déclarer des constants (le mot clé **final** est alors optionnel).

On utilise souvent une interface pour décrire une API : un ensemble de fonctionnalités offertes à l'utilisateur (qui ne se soucie pas de l'implémentation).

## Implémentation d'une interface

Une fois déclarée, une interface peut être utilisée par une autre classe. On dit que cette classe **implémente** l'interface (elle doit pour cela implémenter toutes les méthodes de l'interface). Exemple :

```
public class Rectangle implements Polygone {  
    ...  
    public double perimetre() {  
        return ...;  
    }  
    public double surface() {  
        return ...;  
    }  
}
```

Une classe peut implémenter plusieurs interfaces (les noms d'interfaces sont séparés par des virgules après le **implements**).

# Interfaces

Une interface peut étendre plusieurs autres interfaces (c-à-d inclure les fonctionnalités prévues par d'autres interfaces). On utilise le mot clé **extends** comme pour l'héritage mais ici il peut être multiple (on sépare par des virgules toutes les interfaces que l'on reprend). Lors de l'implémentation il faudra implémenter toutes les méthodes de l'interface et toutes les méthodes des interfaces « héritées ».

On peut se demander quand utiliser une classe abstraite et quand utiliser une interface. Une classe abstraite est plus structurée et peut fournir des implémentations par défaut. Une interface est plus souple puisque une classe peut implémenter plusieurs interfaces (alors qu'elle ne peut hériter que d'une seule classe).

# L'interface Iterable

L'interface **Iterable** du JDK sert à parcourir un ensemble de valeurs indépendamment de l'implémentation. Elle fournit une méthode **iterator** retournant un **Iterator**. Voici sa définition

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

La notation **<T>** correspond aux *generic* apparus en J2SE 5.0 (JDK1.5). Ils servent à paramétriser une classe avec un type **T** (méta-variable). Par exemple une **ArrayList** contenant des **String** s'écrira **ArrayList<String>**. Les generic permettent au compilateur de détecter des erreurs et évitent beaucoup de cast. Les generic ne sont pas obligatoires, si on ne les utilise pas le compilateur Java considérera **T** comme étant un **Object**.

# L'interface Iterator

Que fait-on de l'objet **Iterator<T>** rendu par **iterator()** ?

En fait, **Iterator** est aussi une interface définie comme suit :

```
public interface Iterator<T> {  
    boolean hasNext(); // retourne vrai s'il y a encore un élément.  
    T next(); // retourne le prochain élément (avance)  
    void remove(); // supprime le dernier élément retourné  
}
```

La méthode **hasNext()** permet de savoir s'il reste des valeurs à parcourir.

La méthode **next()** avance dans le parcours et retourne l'élément suivant.

La méthode **remove()** supprime le dernier élément retourné par **next()**.

## Utilisation d'un itérateur

Pour l'affichage de tous les éléments de **c** :

```
Iterator it = c.iterator();
while (it.hasNext())
    System.out.println(it.next());
```

Pour la suppression des éléments de **c** égaux à **x**:

```
Iterator it = c.iterator();
while (it.hasNext()) {
    Object e = it.next();
    if (e.equals(x))
        it.remove();
}
```

## Itérateur et foreach

Lorsqu'un objet **c** fournit un itérateur pour parcourir ses éléments (de type **E**) on peut utiliser une version étendue du **for** (appelée *foreach*) et dont la syntaxe est donné par l'exemple suivant :

```
for(E e : c)
    System.out.println(e);
```

Dans ce cas **c** doit informer qu'il implémenter **iterator()** (qui permet d'obtenir un itérateur). Pour cela il doit implémenter **Iterable**.

```
public class EnsDeValeurs implements Iterable<E> {
    public Iterator<E> iterator() {
        ...
    }
```

## L'interface Comparable

Il est parfois nécessaire que des objets puissent être comparés par une relation d'ordre (pour être triés, pour être stockés dans une structure de données ordonnée,...). Java modélise cela grâce à l'interface **Comparable** définie comme suit :

```
public interface Comparable<T> {  
    public int compareTo(T obj);  
}
```

La méthode **compareTo (obj)** compare l'objet courant (**this**) à l'objet **obj** spécifié en paramètre et retourne :

- 0 si l'objet courant est égal à **obj**
- un nombre < 0 si l'objet courant précède **obj**
- un nombre > 0 si l'objet courant suit **obj**

## L'interface Comparable

Beaucoup de classes prédéfinies implémentent **Comparable** (suivant un ordre usuel) :

- **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double** (ordre numérique)
- **Character** (ordre alphabétique)
- **String** (ordre lexicographique)
- **File** (ordre lexicographique sur le chemin d'accès)
- **Date** (ordre chronologique)

## Exemple d'implémentation de Comparable

On définit un *ordre naturel* entre lecteurs : on compare d'abord sur le nom et en cas d'égalité sur le prénom (ordre « nom puis prénom »).

```
public class Lecteur implements Comparable<Lecteur> {  
    private String nom;  
    private String prenom;  
    ...  
    public int compareTo(Lecteur l) {  
        int cmpNom = nom.compareTo(l.nom);  
        if (cmpNom != 0)  
            return cmpNom;  
  
        return prenom.compareTo(l.prenom);  
    }  
}
```

## Exemple : Tri en utilisant l'ordre naturel

Grâce à notre implémentation de **Comparable** pour définir un ordre naturel on va pouvoir trier un tableau de lecteurs suivant l'ordre naturel « nom puis prénom ».

```
Lecteur []tab = new Lecteur[n];  
// remplissage des n éléments du tableau des lecteurs  
...  
// tri suivant l'ordre naturel « nom puis prénom »  
Arrays.sort(tab);
```

Rq: la classe **Arrays** fournit une méthode de classe **sort(tab)** qui permet de trier un tableau **tab** suivant l'ordre naturel de ses éléments.

## L'interface Comparator

Parfois des données doivent être comparées selon plusieurs relations : parfois, selon le nom, parfois selon le prénom, etc. Dans ce cas, associer à la classe un ordre naturel en lui faisant implémenter l'interface **Comparable** n'est pas suffisant. Java offre pour cela l'interface **Comparator** définie comme :

```
public interface Comparator<T> {  
    public int compare(T obj1, T obj2);  
}
```

La méthode **compare (obj1, obj2)** compare **obj1** et **obj2** et retourne un nombre <, = , > à 0 similairement à **compareTo ()**.

## L'interface Comparator

Définissons un comparateur implémentant un ordre « prénom puis nom ».

```
class PrenomComparator implements Comparator<Lecteur> {  
    public int compare(Lecteur l1, Lecteur l2) {  
        int cmpPrenom = l1.prenom.compareTo(l2.prenom);  
        if (cmpPrenom != 0)  
            return cmpPrenom;  
        return l1.nom.compareTo(l2.nom);  
    }  
}
```

Rq: utiliser une *inner class* pour définir **PrenomComparator** .

## Exemple : tri en utilisant un comparateur

On va pouvoir créer un objet comparateur pour trier un tableau de lecteurs suivant l'ordre « prénom puis nom ».

```
Lecteur [] tab = new Lecteur[n];  
// remplissage des n éléments du tableau des lecteurs  
...  
  
// création d'un comparateur « prenom puis nom »  
Comparator prenomComparator = new PrenomComparator();  
// tri suivant l'ordre « prenom puis nom »  
Arrays.sort(tab, prenomComparator);
```

LE

# FRAMEWORK

# COLLECTION

# Les collections

Une collection est un objet qui représente un groupe d'objets (par exemple l'historique `Vector`). Le framework collections est homogène et normalisé (via les interfaces). Ainsi plusieurs implémentations différentes fournissent les mêmes méthodes et donc sont interchangeables.

L'avantage du framework Collection sont:

- **Réduit l'effort de programmation** : fournit déjà les structures de données + algorithmes. Il n'y a plus qu'à les réutiliser.
- **Réduit l'effort d'apprentissage** : car architecture homogène.
- **Performance** : en général bien programmé. On peut aussi interchanger les implémentations

# Les interfaces

Le JDK définit les **interfaces** suivantes :

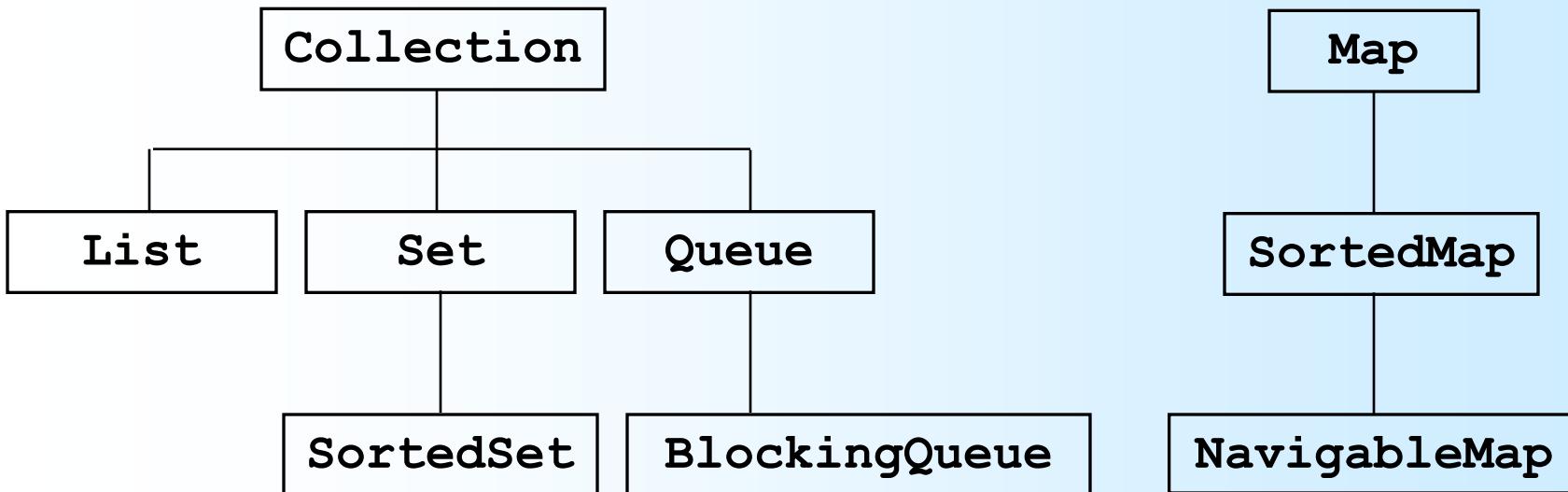
- **Collection** : interface la plus générale.
- **List** : collections ordonnées avec doublons et accès direct.
- **Set** : ensembles (sans doublons).
- **SortedSet** : (étend **Set**) ensembles ordonnés.
- **Queue** : files d'attente (FIFO mais aussi autres).
- **BlockingQueue** : (étend **Queue**) avec blocage (ex: file vide).
- **Map** : gère des couples clé/valeur (pas de doublons de clé).
- **SortedMap** : (étend **Map**) permet d'ordonner les clés.

Ces interfaces s'appuient sur d'autres interfaces déjà vues :

**Iterable / Iterator** : pour énumérer les éléments d'une collection.

**Comparable / Comparator** : permet de comparer 2 objets.

# Les interfaces



Rq: il y a 2 interfaces de plus haut niveau: **Collection** et **Map**.

Une **Map** n'est pas une **Collection** mais elle fournit des méthodes (**values()**, **keySet()**, **entrySet()**) qui retournent une **Collection**.

## Les classes fournies

Les classes qui implémentent ces interfaces ont des noms qui respectent la norme suivante: <*implémentation*><*interface*> :

		<i>Implémentation</i>				
		table hachage	tableau redimens.	arbre équilibré	liste chaînée	table hachage + liste chaînée
<i>interface</i>	<b>Set</b>	<b>HashSet</b>		<b>TreeSet</b>		<b>LinkedHashSet</b>
	<b>List</b>		<b>ArrayList</b>		<b>LinkedList</b>	
	<b>Map</b>	<b>HashMap</b>		<b>TreeMap</b>		<b>LinkedHashMap</b>

Rq: le JDK 1.1 offrait **Vector** et **HashTable** qui continuent à exister mais sont plutôt repris par **ArrayList** et **HashMap**.

# Parcours des collections

Toute collection implémente un itérateur permettant de la parcourir. Ainsi si c'est une collection déclarée puis remplie :

```
Collection<E> c = new ...;  
c.add(...);
```

On peut la parcourir grâce à :

1) l'utilisation implicite de l'itérateur avec un foreach:

```
for(E e : c)  
    System.out.println(e);
```

2) l'utilisation explicite de l'itérateur (cf. l'interface **Iterator**) :

```
Iterator<E> it = c.iterator();  
while (it.hasNext())  
    System.out.println(it.next());
```

# L'interface Collection<E>

**boolean add(E e)** : ajoute l'élément **e** (retourne si OK).

**boolean addAll(Collection c)** : ajoute tous les éléments de **c**.

**void clear()** : supprime tous les éléments.

**boolean contains(E e)** : teste si **e** est dans la collection.

**boolean containsAll(Collection c)** : teste si tous les éléments de **c** sont dans la collection.

**boolean isEmpty()** : retourne si la collection est vide.

**Iterator<E> iterator()** : retourne un itérateur.

**boolean remove(E e)** : supprime un objet donné.

**boolean removeAll(Collection c)** : supprime les éléments de **c**.

**int size()** : retourne le nombre d'éléments de la collection.

**Object[] toArray()** : retourne un tableau de tous les éléments.

## L'interface Set<E>

Une **Set** est une Collection qui ne contient pas d'éléments en double (ce qui correspond à la définition mathématique d'un ensemble).

Les méthodes sont les mêmes que celles de **Collection<E>** sauf que la méthode **add** (et **addAll**) n'ajoutera pas un élément qui s'y trouve déjà (et retournera **false**).

Une **SortedSet** assure en plus que les éléments sont maintenus triés (ordre naturel) et donc rendus dans cet ordre par l'itérateur ou par **toArray()**.

## L'interface List<E>

List<E> ajoute des indices, en plus de Collection on trouve :

`boolean add(E e)` : ajoute l'élément `e` à la fin de la liste.

`void add(int i, E e)` : insère `e` à l'indice `i`.

`boolean addAll(int i, Collection c)` : insère tous les éléments de `c` à partir de l'indice `i`.

`E get(int i)` : retourne l'élément d'indice `i`.

`int indexOf(E e) / lastIndexOf(E e)` : retourne l'indice de la première/dernière occurrence de `e` (ou `-1`).

`E remove(int i)` : supprime l'élément à l'indice `i`.

`E set(int i, E e)` : remplace l'élément d'indice `i` par `e`.

`List<E> subList(int i1, int i2)` : retourne une sous-liste entre `i1` et `i2` (non-compris).

## Exemple : la classe ArrayList<E>

C'est une implémentation de l'interface `List<E>` basée sur des tableaux redimensionnables. L'utilisateur n'a donc pas à se soucier de la taille du tableau, celui-ci est agrandie automatiquement si besoin.

Les éléments sont accessibles via des index (à partir de 0).

Exemple :

```
List<String> tbl = new ArrayList<String>();  
tbl.add("aaa");  
tbl.add("bbb");  
tbl.add(1, "ccc");  
for(String i : tbl)  
    System.out.println(i);
```

aaa  
ccc  
bbb

# L'interface Map<K,V>

Une **Map** est un dictionnaire qui associe à une clé (key) et une valeur. La clé doit être unique. Voici les méthodes fournies:

**void clear()**: supprime tous les éléments.

**boolean containsKey(K k)**: teste si **k** est une clé.

**boolean containsValue(V v)**: teste si **v** est une valeur.

**V get(K k)**: recherche la valeur associée à la clé **k** (ou **null**).

**boolean isEmpty()** : retourne si la Map est vide.

**V put(K k, V v)**: ajoute une entrée, associe à la clé **k** la valeur **v**.

Retourne l'ancienne valeur associée à la clé (ou **null**).

**V remove(K k)** : supprime la clé **k** (et sa valeur). Retourne l'ancienne valeur associée à la clé (ou **null**).

**int size()** : retourne le nombre d'éléments de la Map.

## L'interface Map<K,V> et les vues

Il est possible d'obtenir 3 vues d'une Map (remarque: une **SortedMap** maintient les clés triés).

**Set<K> keySet ()**: retourne toutes les clés.

**Collection<V> values ()**: retourne toutes les valeurs.

**Set<Map.Entry<K,V>> entrySet ()**: retourne toutes les entrées.

La dernière méthode retourne des couples <clé,valeur>. Pour cela le framework définit une interface **Map.Entry<K,V>** qui fournit :

**K getKey ()**: retourne la clé

**V getValue ()**: retourne la valeur

**V setValue (V value)**: change la valeur (renvoie l'ancienne valeur).