Programmation Orientée Objet avec Java

24 octobre 2013

mini-projet

Chapitre I – Introduction

Chapitre I – Introduction

- I. Les notions de la programmation orientée objet
- II. Un premier exemple

Introduction

La programmation structurée est basée sur une décomposition en actions.
La programmation orientée objet travaille avec une décomposition en objet.
Les instructions élémentaires sont les mêmes, mais leur regroupement est différent.
On trouve dans l'approche objet trois principes fondamentaux :

- l'encapsulation : un objet regroupe à la fois ses attributs et ses opérations associées,
- l'indépendance temporelle : le comportement d'un objet est indépendant du contexte dans lequel il est appelé,
- l'indépendance spatiale : les informations relatives à une même entité sont physiquement dans le même module.

Introduction

La programmation structurée est basée sur une décomposition en actions.
La programmation orientée objet travaille avec une décomposition en objet.
Les instructions élémentaires sont les mêmes, mais leur regroupement est différent.
On trouve dans l'approche objet trois principes fondamentaux :

- l'encapsulation : un objet regroupe à la fois ses attributs et ses opérations associées,
- l'indépendance temporelle : le comportement d'un objet est indépendant du contexte dans lequel il est appelé,
- l'indépendance spatiale : les informations relatives à une même entité sont physiquement dans le même module.

Introduction

La programmation structurée est basée sur une décomposition en actions.
La programmation orientée objet travaille avec une décomposition en objet.
Les instructions élémentaires sont les mêmes, mais leur regroupement est différent.
On trouve dans l'approche objet trois principes fondamentaux :

- l'encapsulation : un objet regroupe à la fois ses attributs et ses opérations associées,
- l'indépendance temporelle : le comportement d'un objet est indépendant du contexte dans lequel il est appelé,
- l'indépendance spatiale : les informations relatives à une même entité sont physiquement dans le même module.

La programmation orientée objet apporte de nouvelles notions :

- notion de *classe* : une classe regroupe des objets ayant des propriétés et comportements communs (factorisation des propriétés),
- notion d'*héritage* : une sous-classe est définie à partir d'une classe avec des propriétés supplémentaires ; la sous-classe hérite des propriétés et des opérations de la classe parente,
- notion de *polymorphisme* : permet d'écrire des programmes de même but, donc de même nom, mais dans des contextes différents selon la nature des objets sur lesquels ils portent,
- notion de *liaison dynamique* : capacité à associer le service surchargé correct en fonction de la référence de la classe,

La programmation orientée objet apporte de nouvelles notions :

- notion de *classe* : une classe regroupe des objets ayant des propriétés et comportements communs (factorisation des propriétés),
- notion d'*héritage* : une sous-classe est définie à partir d'une classe avec des propriétés supplémentaires ; la sous-classe hérite des propriétés et des opérations de la classe parente,
- notion de *polymorphisme* : permet d'écrire des programmes de même but, donc de même nom, mais dans des contextes différents selon la nature des objets sur lesquels ils portent,
- notion de *liaison dynamique* : capacité à associer le service surchargé correct en fonction de la référence de la classe,

La programmation orientée objet apporte de nouvelles notions :

- notion de *classe* : une classe regroupe des objets ayant des propriétés et comportements communs (factorisation des propriétés),
- notion d'*héritage* : une sous-classe est définie à partir d'une classe avec des propriétés supplémentaires ; la sous-classe hérite des propriétés et des opérations de la classe parente,
- notion de *polymorphisme* : permet d'écrire des programmes de même but, donc de même nom, mais dans des contextes différents selon la nature des objets sur lesquels ils portent,
- notion de *liaison dynamique* : capacité à associer le service surchargé correct en fonction de la référence de la classe,

La programmation orientée objet apporte de nouvelles notions :

- notion de *classe* : une classe regroupe des objets ayant des propriétés et comportements communs (factorisation des propriétés),
- notion d'*héritage* : une sous-classe est définie à partir d'une classe avec des propriétés supplémentaires ; la sous-classe hérite des propriétés et des opérations de la classe parente,
- notion de *polymorphisme* : permet d'écrire des programmes de même but, donc de même nom, mais dans des contextes différents selon la nature des objets sur lesquels ils portent,
- notion de *liaison dynamique* : capacité à associer le service surchargé correct en fonction de la référence de la classe,

Traduction en Java

- on utilisera des *classes* et des *interfaces*.
- une classe modélise une entité à l'aide d'*attributs* et de *méthodes*.
- un programme construit une *instance* de classes qui est alors appelée *objet*
- les classes peuvent être rangées dans des paquetages

Traduction en Java

- on utilisera des *classes* et des *interfaces*.
- une classe modélise une entité à l'aide d'*attributs* et de *méthodes*.
- un programme construit une *instance* de classes qui est alors appelée *objet*
- les classes peuvent être rangées dans des paquetages

Traduction en Java

- on utilisera des *classes* et des *interfaces*.
- une classe modélise une entité à l'aide d'*attributs* et de *méthodes*.
- un programme construit une *instance* de classes qui est alors appelée *objet*
- les classes peuvent être rangées dans des paquetages

Traduction en Java

- on utilisera des *classes* et des *interfaces*.
- une classe modélise une entité à l'aide d'*attributs* et de *méthodes*.
- un programme construit une *instance* de classes qui est alors appelée *objet*
- les classes peuvent être rangées dans des paquetages

un exemple de programme

Sous linux, on écrit avec un éditeur (gedit ou emacs) le fichier `MaClasse.java`.
(**attention** : Java distingue majuscules et minuscules)

```
public class MaClasse {
public static void main (String[] args) {
System.out.println("affichez ce que vous voulez");}
}// ceci est un commentaire
```

programme

- `class` mot réservé qui indique que l'on commence la description d'une classe, ce mot est nécessairement suivi du nom de la classe : c'est l'entête de la classe (ligne 1)
- { marque le début d'un bloc ; en ligne 2 c'est le début du corps de la classe MaClasse
- une classe contient des attributs et des fonctions ou des méthodes ; ici il n'y a qu'une méthode nommée `main` (ligne 2)
- `public static` sont des *modificateurs*
- `void` indique que la méthode `main` ne renvoie aucune valeur

programme

- l'exécution d'un programme en Java se fait toujours dans une méthode principale qui se nomme toujours `main` ; l'entête de cette méthode est toujours comme dans la ligne 2, on ne peut changer que l'identificateur arg
- `String` est une classe du paquetage `java.lang` ;
- `String [ ]` arg est le paramètre de la méthode `main`, et c'est un tableau de chaînes de caratères
- `System.out.println`("affichez ce que vous voulez") ; est une instruction qui écrit à l'écran *affichez ce que vous voulez* puis passe à la ligne ; `println` est une méthode de l'objet `out` qui appartient à la classe `System`, cet objet sert à écrire dans le fenêtre d'exécution
- toute instruction se termine par un point-virgule

compilation

Pour compiler ce fichier, dans une fenêtre de commande, on exécute la commande

javac MaClasse.java

On peut alors voir dans le répertoire courant un fichier *MaClasse.class*, ce fichier contient le bytecode de la classe *MaClasse*. Ce code est indépendant de la machine (et de son système d'exploitation).
Puis on fait appel à la machine virtuelle Java qui interprète le bytecode au fur et à mesure de l'exécution du programme : on utilise la commande

java MaClasse

qui exécute la méthode `main` contenue dans la classe *MaClasse*.

On prendra l'habitude suivante :
le nom du fichier sera MaClasse.java pour la déclaration `class MaClasse` { ....
D'autre part, on a tout intérêt à séparer les fichiers source des fichiers bytecode .class.
Pour cela on créera deux répertoires jumeaux Source et Class ; on écrira les fichiers source .java dans Source.
Toujours dans le répertoire Source, on compilera par la commande

javac -d ../Class/ MaClasse.java

puis toujours dans le répertoire Source, on exécutera par la commande

java -cp ../Class/ MaClasse

Librairies

Java fournit de nombreuses librairies de classes remplissant des fonctionnalités très diverses : c'est l'API Java (**A**pplication and **P**rogramming **I**nterface /Interface pour la programmation d'applications).
Ces classes sont regroupées par catégories en paquetages (ou «packages»).
Les principaux paquetages :

- java.util : structures de données classiques
- java.io : entrées / sorties
- java.lang : chaînes de caractères, interaction avec l'OS, threads
- java.awt : interfaces graphiques, images et dessins
- java.applet : les applets sur le web
- javax.swing : package récent proposant des composants « légers » pour la création graphiques

JDK

L'environnement de développement fourni par Sun est le JDK (**J**ava **D**evelopment **K**it / Kit de développement Java).
Il contient :

- les classes de base de l'API java (plusieurs centaines),
- la documentation au format HTML (dans le répertoire où est installé le JDK - /jdk1.../docs/api/index.html - ou bien à l'adresse suivante http ://java.sun.com/docs/index.html)
- le compilateur : javac
- la JVM (machine virtuelle) : java
- le visualiseur d'applets : appletviewer
- le générateur de documentation : javadoc

Deux ouvrages :

- en anglais : The Java Programming Language, K. Arnold, J. Gosling et H. David, Addison Wesley, 2000
- en français : Le Langage Java, Irène Charron, Hermès Sciences

Environnement intégré

Il existe de nombreux IDE (Integrated Development Environment) parmi lesquels

- Eclipse
- Netbeans
- JCreator (Windows uniquement)
- ...

Bien sûr il faut apprendre à s'en servir car ils offrent de nombreuses possibilités.
Pour l'instant on fera simple, comme dans l'exemple précédent (sous Linux).

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

Chapitre II – Bases de Java

- I. Structure d'un programme en Java
- II. Les types en Java
- III. Les types primitifs
- IV. La décision
- V. L'itération
- VI. Classes enveloppantes
- VII. Entrées - sorties
- VIII. Construire ses propres fonctions

Structure d'un programme Java

Un programme Java est constitué d'un ou plusieurs fichiers dont le nom est terminé par .java.
La structure générale d'un programma Java est

- Bibliothèques utilisées
- classes
  - attributs (ou variables)
  - méthodes (ou fonctions ou procédures)

Le point d'entrée est toujours la méthode `main` dont l'en-tête est toujours
`public static void main ( String [ ] arg ) {... }`
C'est la méthode automatiquement appelée par Java.

Tout ce qui est délimité par de accolades respectivement ouvrante et fermante sera appelé un *bloc*.
Un programme doit toujours être commenté pour des raisons de lisibilité, d'évolution potentielle . . .
Les commentaires sont

- /∗ sur plusieurs lignes ∗/
- // sur une seule ligne
- / ∗ ∗ sont utilisés pour créer une documentation automatique en html ∗/

Les identificateurs sont composés de suite de lettre ou de chiffre, commencent par une lettre et doivent être différents des mots réservés et des mots-clés.
Par exemple, MaClasse, maClasse, ma_classe, maClasse1,. . .
On respectera cette convention d'écriture :

- tout nom de classe commence avec une majuscule
- tout nom de méthode commence avec une minuscule
- tout nom de variable ou attribut commence avec une minuscule ; par exemple entier
- si le nom de la méthode ou de la variable est composé de plusieurs mots alors le premier caractère des mots suivants le premier mot sont en majuscule ; par exemple changerDePlace() , monEntierAMoiEtAPersonneDAutre.

Les types en Java

Toutes les données manipulées en Java doivent être **typées**.
En Java il y a deux catégories de type, les types **primitifs** et les autres.
Les types primitifs comprennent les types : booléen, caractère, entier ou réel.
Les autres types sont les types tableau ou les types objet.
Par exemple, `String` est un type objet, `int[ ]` est un type de tableaux d'entiers.
Pour créer de nouveaux types on peut construire des tableaux ou définir des *classes*.

Par exemple, on définit une classe `Personne` dans le but de définir une classe `Carnet` qui contiendra des `Personne` dans un tableau.

```
class Personne {
String nom;
integer age;
....}
class Carnet {
Personne[] liste;
...
```

En Java la manipulation des données diffère selon leur type.

Règle : LES DONNÉES DE TYPE PRIMITIF SONT MANIPULÉES PAR VALEUR ET LES DONNÉES DES AUTRES TYPES SONT MANIPULÉES PAR RÉFÉRENCE

La référence d'une donnée est l'adresse en mémoire de cette donnée.

La référence est typée par le type de la donnée dont elle est l'adresse mémoire.

Selon la règle énoncée une variable de type primitif contient une valeur de ce type et une variable de type non primitif contient l'adresse mémoire où est stockée la valeur de type.

Exemple

```
public class Essai {
public static void main(String[] args) {
byte[] t={1,2,3};
byte[] s;
short x=1;
short y;
s=t;
System.out.println("tableau 1 "+ t);
// tableau 1 [B@6e1408
//              adresse en héxadécimal (base 16)
System.out.println("tableau 2 "+ s);
// tableau 2 [B@6e1408
t[0]=-1;
System.out.println("tableau 1, 1ère valeur "+ t[0]);
// tableau 1 [B@6e1408 -1
System.out.println("tableau 2, 1ère valeur "+ s[0]);
// tableau 2 [@6e1408 -1
System.out.println("x="+x);// x=1
y=x; x=2;
System.out.println("x="+x);// x=2
System.out.println("y="+y);// y= 1
}
}
```

Affectation des types non primitifs

La manipulation des données de type non primitif est limitée à la *comparaison* et l'*affectation*.
Dans ce dernier cas c'est l'adresse mémoire qui est dupliquée et non pas les données référencées.

Remarques :

- la syntaxe de l'affection se fait est le symbole $=$
- la variable à gauche reçoit la valeur de l'expression à droite

Les types primitifs

Les variables de type primitif

- `boolean`
- `int`
- `float`
- `char`

sont manipulées par **valeur**.

le type booléen

Le type `boolean` a deux valeurs `true` et `false`.
Il est codée sur 1 bit.
On verra plus tard les opérateurs logiques.

le type caractère

Le type `char` est codé sur 2 octets selon le code UNICODE 2.1.
Les valeurs de type `char` sont écrites entre apostrophes soit directement soit avec le code unicode sous la forme \uhhhh où hhhh est un chiffre héxadécimal.
Par exemple \u03C6 correspond à φ.
(voir `http://www.unicode.org` pour les caractères en Unicode)

les types entier

Selon la dimension du codage, on distingue les types

- `byte` : 1 octet, intervalle $[-128, 127]$
- `short` : 2 octets, intervalle $[-32768, 32767]$
- `int` : 4 octets, intervalle $[-2147483648, 2147483647]$
- `long` : 8 octets, intervalle $[-9.10^{18}, 9.10^{18}]$ environ

écriture des entiers

Les valeurs entières peuvent s'écrire

- en base 10 : par exemple 12
- en base 8 (octal) indiquée par un 0 en préfixe : par exemple 014
- en base 16 (héxadécimal) indiquée par 0x en préfixe : par exemple 0xc
- les valeurs de type `long` sont suffixées par l ou L.

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Opérations sur les entiers

- $+, -$: signe (arité 1)
- $+, -, *$: addition, soustraction, multiplication
- $/$: division entière
- $x\%y$: modulo (reste de la division entière de x par y) respectant la règle suivante $(x/y) * y + x\%y$ est égale à x

remarque : les calculs sur ces variables se font modulo la portée de leur type ; par exemple pour le type `byte` de portée 256, $100 + 100$ est égal à -56.

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++;$ équivaut à $i = i + 1$
- $x = ++i;$ équivaut à $i = i + 1; x = i;$
- $x = i++;$ équivaut à $x = i; i = i + 1;$
- $i--;$ équivaut à $i = i - 1$
- $x = --i;$ équivaut à $i = i - 1; x = i;$
- $x = i--;$ équivaut à $x = i; i = i - 1;$

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++;$ équivaut à $i = i + 1$
- $x = ++i;$ équivaut à $i = i + 1; x = i;$
- $x = i++;$ équivaut à $x = i; i = i + 1;$
- $i--;$ équivaut à $i = i - 1$
- $x = --i;$ équivaut à $i = i - 1; x = i;$
- $x = i--;$ équivaut à $x = i; i = i - 1;$

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++$; équivaut à $i = i + 1$
- $x = ++i$; équivaut à $i = i + 1; x = i;$
- $x = i++$; équivaut à $x = i; i = i + 1;$
- $i--$; équivaut à $i = i - 1$
- $x = --i$; équivaut à $i = i - 1; x = i;$
- $x = i--$; équivaut à $x = i; i = i - 1;$

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++;$ équivaut à $i = i + 1$
- $x = ++i;$ équivaut à $i = i + 1; x = i;$
- $x = i++;$ équivaut à $x = i; i = i + 1;$
- $i--;$ équivaut à $i = i - 1$
- $x = --i;$ équivaut à $i = i - 1; x = i;$
- $x = i--;$ équivaut à $x = i; i = i - 1;$

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++;$ équivaut à $i = i + 1$
- $x = ++i;$ équivaut à $i = i + 1; x = i;$
- $x = i++;$ équivaut à $x = i; i = i + 1;$
- $i--;$ équivaut à $i = i - 1$
- $x = --i;$ équivaut à $i = i - 1; x = i;$
- $x = i--;$ équivaut à $x = i; i = i - 1;$

Opérations sur les entiers

Il existe des opérateurs permettant une incrémentation plus facile à écrire.

- $i++;$ équivaut à $i = i + 1$
- $x = ++i;$ équivaut à $i = i + 1; x = i;$
- $x = i++;$ équivaut à $x = i; i = i + 1;$
- $i--;$ équivaut à $i = i - 1$
- $x = --i;$ équivaut à $i = i - 1; x = i;$
- $x = i--;$ équivaut à $x = i; i = i - 1;$

Opérateurs de comparaison et opérateurs logiques

- == teste l'égalité
- ! = teste la différence
- <=, <, >, => testent l'ordre
- && est l'opérateur booléen ET,
- || est l'opérateur booléen OU,
- ! est l'opérateur booléen de négation.

L'évaluation d'une expression booléenne se fait de gauche à droite et est stoppée dès que le résultat est connu.
En effet `false` && ... est faux quelque soit la valeur qui suit `false`.
De même, `true` || ... est vrai quelque soit la valeur qui suit `true`.

Les types réels

Selon la dimension du codage, on distingue les types

- `float` : simple précision sur 4 octets
- `double` : double précision sur 8 octets

écriture des flottants

un réel comporte toujours un point (notation anglo-saxonne de la virgule) même s'il a une valeur entière.
Par exemple 1.f vaut 1 ; c'est un flottant et sera affiché 1.0.
Une valeur constante de type `float` est toujours suffixée par f ; par exemple, 10.2f est de type `float`.
Une valeur constante de type `double` n'a pas de rajout ; par exemple, 10.2 est de type `double`.
On peut utiliser une notation scientifique ; par exemple, 1.02e1f de type `float` ou .102e2 de type `double`.

Opérations sur les réels

- $+, -$: signe (arité 1)
- $+, -, *, /$: addition, soustraction, multiplication, division

D'autres opérations sont disponibles dans la classe `java.lang.Math` qu'il faut importer si on veut l'utiliser.
On y trouve par exemple

- les fonctions trigonométriques (argument en radian) : `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- les fonctions logarithmiques et exponentielles : `log`, `exp`, `sqrt`, `pow`
- arrondi : `round`, `ceil`, `floor`
- valeur absolue : `abs`

Pour toutes ces fonctions l'argument est de type `double`.

Exemple

```
import java.lang.Math;
public class Essai2 {
public static void main(String[] args) {
System.out.println(Math.sqrt(4.0)) ;//2.0
System.out.println(Math.sin(Math.PI/2)) ;//1.0
}}
```

voir l'API pour la classe Math.

Conversions implicites entre types

- `byte` $\rightsquigarrow$ `short`
- `short` $\rightsquigarrow$ `int`
- `char` $\rightsquigarrow$ `int`
- `int` $\rightsquigarrow$ `float`
- `float` $\rightsquigarrow$ `double`

Par exemple si x est de type `short` et y de type `int`, l'affectation y=x transforme automatiquement la valeur de x en une valeur de type `int` pour l'affecter à y.
Pour le type `char`, une valeur de ce type a un code unicode \uhhhh ; lorsque cette valeur est affectée à une variable i de type int hhhh est calculé en décimal et affecté à i.
Par exemple,

```
i = '2'; System.out.println(i);
```

affiche 50 car le caractère '2' a le code \u0032.

Conversions explicites entre types

On peut explicitement faire les conversions suivantes

- `byte` $\rightsquigarrow$ `char`
- `short` $\rightsquigarrow$ `char`
- `short` $\rightsquigarrow$ `byte`
- `int` $\rightsquigarrow$ `char`
- `int` $\rightsquigarrow$ `short`
- `char` $\rightsquigarrow$ `byte`
- `char` $\rightsquigarrow$ `short`
- `double` $\rightsquigarrow$ `float`
- `float` $\rightsquigarrow$ `int`

La conversion se fait de la façon suivante : (nouveau type) (expression), où expression est du type d'origine.

Conversions explicites entre types

Attention : la conversion s'effectue après évaluation de l'expression dans son type d'origine.
Par exemple

```
short s= -140;
byte b=(byte) s;//b vaut 116 car -140 + 256 = 116
```

Constantes

On peut nommer des constantes de la façon suivante :

```
static final int max = 100;
```

`final` signifie que l'on ne pourra pas changer sa valeur.
L'ordre entre `final` et `static` n'a pas d'importance.

La décision

On va voir deux instructions de décision `if` et `switch`

syntaxe de if

`if (expression) {SuiteDInstructions1;} else {SuiteDIinstructions2;}`
dans laquelle `expression` est une expression booléenne.
Remarques :

- `else` n'est pas obligatoire,
- Si `SuiteDInstructions` ne contient qu'une unique instruction alors les accolades ne sont pas obligatoires.

syntaxe de if

règle : SANS ACCOLADES, `ELSE` SE RAPPORTE TOUJOURS AU `IF` PRÉCÉDENT LE PLUS PROCHE.

Selon cette règle, dans l'instruction suivante

`if (expression1) if (expression2) instruction1; else instruction2;`

`instruction2` s'exécute lorsque `expression1` est vraie et `expression2` fausse. Rien n'est prévue si `expression1` est fausse.

syntaxe de if

Si `SuiteDInstructions1` et `SuiteDInstructions2` sont chacune une affectation à **la même variable** x alors on peut écrire

```
x = expression1? expression2 : expression3;
```

avec `expression1` de type booléen, `expression2` et `expression3` de même type que x.

`expression2` est affecté à x si `expression1` est évaluée à `true`
`expression3` est affecté à x si `expression1` est évaluée à `false`

syntaxe de switch

L'instruction `switch` permet de faire plusieurs tests sur la valeur d'une même variable (uniquement de type `byte, short, int, char`).
La syntaxe est la suivante :

```
switch (variable){
case valeur1 : SuiteDinstructions1;break;
...
case valeurk : SuiteDinstructionsk;break;
default : SuiteDinstructions;break;
}
```

Remarque : on peut écrire plusieurs case séparés par : pour une seule suite d'instructions

```
case valeur1 : case valeur2 : case valeur3 :
SuiteDinstructions;break;
```

sémantique de switch

Pour chaque `case` la valeur de variable est évaluée puis comparée à la valeur du `case` ; si elles sont égales la suite d'instructions correspondante est exécutée.
Le mot clef `default` indique la suite d'instructions à exécuter lorsqu'aucune des valeurs des `case` ne sont égales à la valeur de la variable.
Attention : si on omet `break` à la fin des instructions, alors toutes lesinstructions suivant la première exécutée seront à leur tour exécutées sans tenir compte des valeurs.

L'itération

On va voir les instructions

- `for`
- `while`
- `do while`

l'instruction for

La syntaxe est la suivante

```
for (compteur; test; modification){
  SuiteDInstructions; }
```

- `compteur` est la variable qui sert de contrôle à la boucle (avec l'initialisation du compteur et/ou sa déclaration le cas échéant)
- `test` est la condition que doit vérifier le compteur pour **continuer** la boucle `for`,
- `modification` est une instruction qui modifie la valeur du compteur de façon à ce que test devienne faux (sinon boucle infinie),
- si `SuiteDInstructions` ne contient qu'une instruction les accolades ne sont pas indispensables.

for

`for`(i=0;i<10; i++) effectue 10 itérations pour i variant de 0 à 9.
`for`(i=1;i<10; i++) effectue 9 itérations pour i variant de 1 à 9.
`for`(i=1;i<=10; i++) effectue 10 itérations pour i variant de 1 à 10.
`for`(`int` j=10;j>=1; j−−) effectue 10 itérations pour j variant de 10 à 1.
Remarques :

- les boucles `for` peuvent être imbriquées
- le compteur peut être utilisé dans la suite d'instructions
- si compteur est déclaré dans la boucle `for` sa portée est limitée à la boucle (il sera invisible en dehors).

l'instruction while

La syntaxe est la suivante

```
while (expressionBooléenne) {
SuiteDInstructions; }
```

`expressionBooléenne` est évaluée et tant qu'elle est vraie alors `SuiteDInstructions` est exécutée.
Il faut bien sûr que cette suite d'instructions fasse évoluer une ou des variables de l'expression booléenne de façon à ce qu'elle devienne fausse sinon on a un risque de boucle infinie.

l'instruction do while

La syntaxe est la suivante

```
do { SuiteDInstructions; }
while (expressionBooléenne);
```

`SuiteDInstructions` est exécutée et tant que `expressionBooléenne` est vraie on répète cette exécution. De même on doit s'assurer que l'expression booléenne devienne fausse au cours de l'exécution de la suite d'instructions.

différence en while et do while

L'expression booléenne n'est pas évaluée au même moment
Avec `while` l'expression booléenne est évaluée avant l'exécution des instructions.
Avec `do while` l'expression booléenne est évaluée après l'exécution des instructions.
Donc , la suite d'instructions est toujours exécutée au moins une fois avec `do while` .

Dérouter une itération

Il est possible

- d'interrompre une boucle par l'instruction `break`
- de sauter une partie des instructions de la boucle par l'instruction `continue`.

`break` l'itération est stoppée et les instructions suivant cette itération sont exécutées

`continue` une boucle `while (expression booléenne){}` ou `do {}while (expression booléenne)` reprend au test de l'expression booléenne

`continue` une boucle `for` passe à l'itération suivante

Classes enveloppantes

Il est possible de considérer les types primitifs comme des types objet par l'intermédiaire des « classes enveloppantes » qui existent pour chaque type primitif :

- java.lang.Byte $\rightsquigarrow$ byte
- ...
- java.lang.Float $\rightsquigarrow$ float
- java.lang.Boolean $\rightsquigarrow$ boolean
- java.lang.Character $\rightsquigarrow$ character

Ces classes possèdent des méthodes permettant le traitement relatif au type primitif associé.

exemple

```
public class TestEnveloppante {
public static void main( String [] args ) {
int n = Integer.parseInt("213");
System.out.println(n+3);// affiche 216
}
```

Sorties

On peut afficher la valeur d'une variable x avec

```
System.out.println(x);
```

qui dans ce cas va à la ligne après affichage.
La commande

```
System.out.print(x);
```

affiche x et laisse le curseur à coté de la valeur de x.

Sorties

Pour afficher un texte il faut le mettre entre guillemets.
On peut afficher les contenus de plusieurs variables et du texte en les séparant par un $+$.
On dit que le symbole $+$ est un symbole de *concaténation*.

En ligne de commande

Une première possibilité est de profiter de la structure de la méthode `main`.
Elle a comme argument un tableau de chaînes de caractères. On peut alors fournir des valeurs pour ces paramètres en ligne de commande.
Par exemple

```
public class Argument {
public static void main(String[] args) {
int x;
x =  Integer.parseInt(arg[0]);
System.out.println("L'entier vaut " + x);
}
```

Entrées

Sinon dans un environnement non graphique on utilisera la classe `Scanner` du package `java.util` (à partir de Java 1.5 seulement) pour réaliser des saisies au clavier.
Grâce à cette classe on peut saisir des valeurs numériques `byte`, `short`, `int`, `long`, `float`, `double` et des valeurs caractères ou chaînes de caractères.
On commence tout d'abord avant la définition de la classe par importer la classe Scanner

```
import java.util.Scanner;
```

Puis on crée un objet de type Scanner de la façon suivante

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

la classe Scanner

On utilise une méthode de la classe `Scanner` qui permet de lire un entier, un réel ou une chaîne de caractères.

```
int i;
i=saisieClavier.nextInt();
```

Pour les valeurs numériques les méthodes sont `nextByte`, `nextShort`, `nextInt`, `nextLong`, `nextFloat`, `nextDouble`.
Pour une chaîne de caractères String on utilise la méthode saisieClavier.`next`() qui saisit la suite de caractères jusqu'au caractère espace qui marque la fin de la saisie.
Pour saisir une phrase composée de plusieurs mots on utilise la méthode saisieClavier.`nextLine`().
Pour saisir un caractère on utilise la méthode saisieClavier.`next`().`charAt`(0) qui saisit le premier caractère entré.

Exemple

```
import java.util.Scanner;
public class EssaiPgcd {
 // fct qui renvoie le pgcd de a et b
static int pgcd(int a,int b){
while (a != b) {
if (a<b) b=b-a ; else a-=b ;}
return a ;}
// méthode main
public static void main ( String [ ] arg ) {
int x ; int y ;
Scanner saisieClavier = new Scanner(System.in) ;
System.out.println("entrez 2 entiers positifs");
System.out.println("sinon entrez (0 pour arreter)");
x=saisieClavier.nextInt() ;
while (x > 0) {
y=saisieClavier.nextInt() ;
if (y <= 0) break; else {
System.out.println("le pgcd de "+x+" et "+y+
                          " est " +pgcd(x,y));
System.out.println("entrer deux entiers positifs
                            (0 pour arreter)");
x=saisieClavier.nextInt() ; }
} } }
```

Fonctions

On distinguera les sous-programmes s'appliquant à des données de types primitifs et les autres. Pour le premier cas on parlera toujours de fonctions mais de méthodes pour les types non primitifs.

Syntaxe

Pour déclarer une fonction on écrira pour l'instant si la fonction renvoie un résultat

```
public static typeResultat nomDeLaFonction (listeDeParamètres){
  SuiteDInstructions;
  }
```

ou bien si la fonction ne renvoie pas de résultat

```
public static void nomDeLaFonction (listeDeParamètres){
  SuiteDInstructions;
  }
```

Syntaxe

- `typeResultat` est le type de la valeur qui est retournée par la fonction,
- nomDeLaFonction est le nom de la fonction lequel sera utilisé pour l'appel de la fonction,
- listeDeParamètres est la liste des paramètres avec lesquels la fonction sera appelée,
- (listeDeParamètres) est de la forme (`type1` identPara1, `type2` identPara2, ...),
- ces paramètres sont dits *formels*,
- listeDeParamètres peut être vide,
- si la fonction retourne une valeur SuiteDInstructions doit contenir une instruction `return` valeur ;
- une telle fonction ne renvoie qu'**une seule valeur**,
- l'instruction `return` valeur ; marque la fin de la fin fonction, la méthode appelante reprend le contrôle,

void

`void` indique que la fonction ne retourne aucune valeur

Remarque : une instruction

```
return;
```

peut être utilisée lorsque la fonction ne retourne aucune valeur pour, par exemple, sortir d'une instruction conditionnelle.

Visibilité

Les instructions figurant dans la fonction peuvent utiliser

- les paramètres
- des variables propres à la fonction que l'on dira locales à la fonction.

Règle : TOUTE VARIABLE DÉCLARÉE À L'INTÉRIEUR D'UNE FONCTION N'EST VISIBLE QUE DANS CETTE FONCTION.
Cette règle s'applique aussi aux méthodes et donc à la méthode principale `main`.
Pour une fonction, ses variables locales n'existent que le temps de l'exécution de la fonction et ne peuvent pas être utilisées par d'autres fonctions parce que non visibles.

appel

Une fois la fonction définie elle peut être utilisée par un autre fonction ou méthode.
La syntaxe à respecter est le nom de la fonction suivie entre parenthèses d'une liste de paramètres en même nombre et même type respectif que dans la définition de la fonction.
Les variables sur lesquelles est *appliquée* la fonction sont dits paramètres réels ou effectifs.
Si la fonction renvoie un résultat elle sera utilisée comme une variable du type renvoyé : affectation, test, dans une autre fonction ou méthode ....
Par exemple,

```
z=pgcd(x,y);
```

passage de paramètres

Règle : en Java les arguments d'une fonction sont passés par valeur, c'est-à-dire les valeurs des paramètres réels sont copiés au moment de l'appel.

Fonction récursive

Une fonction peut appeler une autre fonction comme `estPremier` utilisant `pgcd`.
Elle peut s'appeler elle-même : on parle alors de fonction récursive.
Par exemple, le calcul de la fonction factorielle s'écrira :

```
import java.util.Scanner ;
public class EssaiFact {
static int Fact(int n){
if (n<=1)
return 1 ;
else return n*Fact(n-1) ;
}
public static void main ( String [ ] arg ) {
int x ;
Scanner saisieClavier = new Scanner(System.in) ;
System.out.print("entrer un entier: x=");
x=saisieClavier.nextInt() ;
System.out.println("x! = "+ Fact(x));
} }
```

Exercices

1. Ecrire un programme qui saisit un entier n et qui affiche le terme de rang n de la suite de Fibonacci.
2. Ecrire un programme qui saisit un entier n et qui affiche le nombre d'itérations de la suite de Syracuse pour atteindre 1.
3. Ecrire un programme qui saisit un entier seuil et qui affiche le rang k du premier terme de la suite de Fibonacci qui dépasse seuil.

Chapitre III – Les tableaux simples

Chapitre III – Les tableaux simples

Chapitre III – Les tableaux simples

Chapitre III – Les tableaux simples

- I. Déclaration
- II. Construction
- III. Initialisation et Manipulation
- IV. Tableaux à plusieurs dimensions
- V. Dépassement de bornes

Chapitre III – Les tableaux simples

- I. Déclaration
- II. Construction
- III. Initialisation et Manipulation
- IV. Tableaux à plusieurs dimensions
- V. Dépassement de bornes

tableau

Un tableau est une collection de valeurs qui sont tous de même type (primitif ou objet).
En Java on distingue la déclaration, la construction et l'initialisation d'un tableau.
Dans ce chapitre on ne verra que des tableaux d'entiers.

Déclaration

La syntaxe de la déclaration d'un tableau nomTableau contenant des éléments de type `typeElement` est :

`typeElement` [ ] nomTableau ;

Construction

La syntaxe de construction d'un tableau nomTableau déjà déclaré est :

nomTableau = `new typeElement [nbreElement];`

`nbreElement` correspond au nombre d'éléments de nomTableau. C'est *une valeur de type entier qui doit pouvoir être évaluée au moment de l'exécution*.

Cette construction correspond à l'allocation mémoire des `nbreElement` espaces mémoire pour contenir les éléments du tableau. Une fois cette dimension fixée *elle ne peut pas être modifiée*.

A ce moment, nomTableau est une référence à ces espaces mémoire ; de plus chacun de ces espaces mémoire est initialisée selon `typeElement` : soit à `null` pour un type objet, soit 0 pour un type entier et 0.0 pour un type flottant.

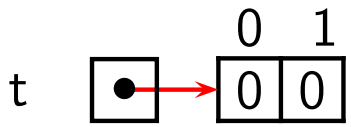
numérotation

Un tableau est **toujours numéroté depuis 0**.
Tout tableau a un attribut `length` permettant de connaître sa longueur soit le nombre d'éléments qu'il contient. Si `tab` est l'identificateur du tableau `tab.length` sera le nombre d'éléments de `tab`.

Exemple

```
t = new int[2];
```

peut s'illustrer ainsi



Initialisation et Manipulation

On peut regrouper ces différentes étapes de plusieurs façons :

- `int[ ] t ={1,2,3};` : cela définit directement un tableau de 3 entiers contenant 1, 2, 3.
-
  - `char[] v ;`
  - `v = new char[2];`
  - `v[0]='o'; v[1]='h';`

Dans l'exemple suivant, on va utiliser la classe `Random` de `java.util.` pour créer une méthode qui initialise un tableau d'entiers par des valeurs engendrées au hasard.

Exemple

```
static void initTableau(int [] t){
Random randomGenerator = new Random() ;
int randomInt = randomGenerator.nextInt(100);
for (int i=0 ;i<t.length ; i++){
t[i]=randomInt ;//t[i] appartient à [0,99]
randomInt = randomGenerator.nextInt(100); }
}
```

Tableaux à plusieurs dimensions

Il est possible de manipuler des tableaux de tableaux. Par exemple,

```
int [][] tab ;
t =new int [2][3];
for (int i=0 ;i<tab.length ; i++){
for (int j=0 ;j<tab[i].length ; j++){ t[i][j] = i+j ;
}}
for (int i=0 ;i<t.length ; i++){
for (int j=0 ;j<tab[i].length ; j++){
System.out.print(tab[i][j] + " ");}
System.out.println() ;
}
```

donne comme affichage

0 1 2

1 2 3

On peut aussi avoir des lignes de taille différente par exemple

```
int [][] tab ;
t =new int [2][];
for (int i=0 ;i<t.length ; i++){
t[i] = new t[i+1] ;
for (int j=0 ;j<=i ; j++){ t[i][j] = i+j ;
}}
for (int i=0 ;i<t.length ; i++){
for (int j=0 ;j<t[i].length ; j++){
System.out.print(t[i][j] + " ");}
System.out.println() ;
}
```

donne comme affichage

```
0
1 2
2 3 4
```

Dépassement de bornes

Comme dans tout langage, en Java, faire appel à un élément t[i] où i n'appartient pas à l'intervalle [0. . . t.length-1] provoque une erreur à l'exécution.
Pour ne pas interrompre un programme dans lequel une telle erreur puisse être commise on peut utiliser un mécanisme de gestion des exceptions ; c'est donc un premier exemple d'introduction à cette notion.

Exemple d'exception

```
int [] t ;
t =new int [3];
for (int i=0 ;i<t.length ; i++){
t[i] = i+1;}
for (int i=0 ;i<t.length ; i++){
System.out.print(t[i]);}
try { t[3] = 4 ;
}
catch (ArrayIndexOutOfBoundsException) {
System.out.println("erreur interceptée ");
}
```

donne l'affichage suivant

1 2 3

erreur interceptée

Chapitre IV – Classes, attributs, méthodes

Chapitre IV – Classes, attributs, méthodes

Chapitre IV – Classes, attributs, méthodes

Chapitre IV – Classes, attributs, méthodes

Chapitre IV – Classes, attributs, méthodes

Chapitre IV – Classes, attributs, méthodes

- I. Un premier exemple
- II. Attributs et méthodes : généralités
- III. Méthodes
- IV. Contrôles d'accès
- V. Constructeurs
- VI. Disparition d'un objet
- VII. La classe String
- VIII. La classe `StringBuffer`

Chapitre IV – Classes, attributs, méthodes

Chapitre IV – Classes, attributs, méthodes

- I. Un premier exemple
- II. Attributs et méthodes : généralités
- III. Méthodes
- IV. Contrôles d'accès
- V. Constructeurs
- VI. Disparition d'un objet
- VII. La classe String
- VIII. La classe `StringBuffer`

Chapitre IV – Classes, attributs, méthodes

- I. Un premier exemple
- II. Attributs et méthodes : généralités
- III. Méthodes
- IV. Contrôles d'accès
- V. Constructeurs
- VI. Disparition d'un objet
- VII. La classe String
- VIII. La classe `StringBuffer`

Généralités

On s'intéresse d'abord à aux entités que l'on va manipuler avant de s'intéresser à la façon de les manipuler.
Une classe est un bloc de base d'un programme Java.
Définir une classe c'est construire un type de données structuré grâce à ses *attributs* et lui adjoindre des *méthodes* permettant sa manipulation (initialisation, modification, consultation, suppression . . .) par l'intermédiaire de ses *attributs*.
Une classe est donc un modèle pour un ensemble de données.
On trouve donc la notion d'*encapsulation* : sous une même entité - la *classe* - sont regroupées les données - les *attributs* - et les moyens - les *méthodes* - de les manipuler.
Un objet est *une instance* d'une classe - une « application concrète » de cette classe pour un usage particulier dans un programme.

Généralités

D'une façon générale, toute classe est construite ainsi :

```
class nomClasse {
// les attributs
// les méthodes
// des classes internes éventuelles
```

Un attribut est une variable mais toute variable n'est pas un attribut puisqu'elle peut être locale à une méthode.
Parmi les méthodes il y a des *constructeurs* de même nom que la classe, **sans type de retour** : ils servent à créer des objets.

Généralités

D'une façon générale, toute classe est construite ainsi :

```
class nomClasse {
// les attributs
// les méthodes
// des classes internes éventuelles
```

Un attribut est une variable mais toute variable n'est pas un attribut puisqu'elle peut être locale à une méthode.

Parmi les méthodes il y a des *constructeurs* de même nom que la classe, **sans type de retour** : ils servent à créer des objets.

Généralités

D'une façon générale, toute classe est construite ainsi :

```
class nomClasse {
// les attributs
// les méthodes
// des classes internes éventuelles
```

Un attribut est une variable mais toute variable n'est pas un attribut puisqu'elle peut être locale à une méthode.
Parmi les méthodes il y a des *constructeurs* de même nom que la classe, **sans type de retour** : ils servent à créer des objets.

Un premier exemple

```
public class Point{
 private float abs;
 private float ord;
 //constructeurs
public Point(float xa, float xb){
abs=xa;ord=xb;}

public Point(){}

//méthodes en écriture
public void setAbs(float x){
this.abs=x;
}
public void setOrd(float y){
 this.ord=y;
}
//méthodes en lecture
public float getAbs(){
return this.abs;
}
public float getOrd(){
return this.ord;
}
```

suite

```
// méthode statique
public static boolean sontAlignes(Point A, Point B, Point C){
return ((B.ord–A.ord)(C.abs–A.abs)==
(C.ord–A.ord)(B.abs–A.abs));
}
// méthode d'instance
public  void afficher(){
System.out.println("abscisse =" + this.abs +
", ordonnée=" + this.ord);
}}
```

this | Surcharge | Accesseurs | constructeur

Attributs et méthodes : généralités

Dans la définition d'une classe il y a deux catégories d'attributs et de méthodes :

- les attributs et méthodes de classe ou statique ; ils sont précédés du modificateur `static`
- les attributs et méthodes d'instance

A l'intérieur de la définition de la classe les attributs et méthodes statiques de cette classe seront accessibles directement sans préfixer leur nom par le nom de la classe. Par exemple sontAlignes.
Dans un programme dès qu'une classe est invoquée, le bytecode de cette classe est chargée en mémoire et les attributs et méthodes statiques n'étant liés qu'à la classe et non à une instance sont immédiatement disponibles. Même si plusieurs instances de cette classe existent **il n'y aura qu'une version d'un attribut de classe**.

Attributs

Si une variable est déclarée `static`, tous les objets de la classe partageront cette variable. Donc, toute modification d'une variable statique dans un objet quelconque de la classe est répercutée dans tous les objets de la classe.

Exemple

```
public class UneClasse {
int x;
static int y ;
}
public class EssaiUneClasse{
public static void main ( String [ ] arg ) {
UneClasse c= new UneClasse();
UneClasse cc= new UneClasse();
c.x=1;
c.y=2;
cc.x=2*c.x;
cc.y=1;
System.out.print(c.x + " ,");
System.out.println(c.y);
System.out.print(cc.x + " ,");
System.out.println(cc.y);
}}
```

L'affichage donne :

1,1
2,1

- les attributs d'une classe, s'ils ne sont pas initialisés, se voient affecter automatiquement une valeur par défaut. Cette valeur vaut : 0 pour les variables numériques, `false` pour les booléens, et `null` pour les tableaux et types d'objet
- le nom véritable d'un attribut de classe y pour la classe UneClasse est UneClasse.y

Portée des variables

Règle : LES VARIABLES SONT CONNUES ET SEULEMENT CONNUES À L'INTÉRIEUR DU BLOC DANS LEQUEL ELLES SONT DÉCLARÉES.
On parle de variable locale lorsqu'une variable est paramètre d'une méthode ou bien déclarée dans un bloc d'exécution comme dans une méthode par exemple.
Une variable locale n'est pas un attribut et *doit être explicitement initialisée* avant d'être utilisée.
En cas de conflit de nom entre des variables locales et des variables globales, c'est toujours la variable la plus locale qui est considérée comme étant la variable désignée par cette partie du programme. Le mieux est d'éviter ce genre de conflit.

Syntaxe

La syntaxe est la suivant
⟨ modificateur ⟩ ⟨ type-retour⟩ ⟨ nom⟩ (⟨ liste-param⟩) {bloc }
avec

- modificateur = public, static . . .
- type-retour = type de la valeur renvoyée ou void
- liste-param = couples de `type identificateur` séparés par des virgules.

Java n'implémente qu'un seul mode de passage des paramètres à une méthode : **le passage par valeur**.

Sémantique

L'argument passé à une méthode ne peut être modifié ; si l'argument est une instance, c'est sa référence qui est passée par valeur. Ainsi, le contenu de l'objet peut être modifié, mais pas la référence elle-même.

Méthode statique

Si une méthode est déclarée static, tous les objets de la classe partageront cette méthode.
Régle : une méthode statique n'a pas directement accès aux variables non statiques.

Exemple

```
public class EssaiUneMethodeStat {
int x;
public static void main ( String [ ] arg ) {
System.out.print("la valeur x vaut " + x);
}}
```

cela provoque une erreur de compilation car `main` est une méthode statique et `x` est un attribut d'instance.

Méthode d'instance

Une méthode qui n'est pas déclarée `static` est toujours utilisée *en référence à un objet*. On parle de méthode *d'objet ou d'instance*.

Appel d'une méthode

- pour appeler une méthode statique : on écrit `nomClasse.nomMethode` si `nomMethode` est une méthode statique de la classe `nomClasse`
- pour appeler une méthode d'objet : on écrit `nomObjet.nomDeMethode(...)`

Par exemple, la classe `System` contient un attribut statique `out` de type `PrintStream`, classe contenant la méthode d'objet `println()`. Donc l'objet `System.out` appelle une méthode d'objet de nom `println()`.

Appel d'une méthode

- pour appeler une méthode statique : on écrit `nomClasse.nomMethode` si `nomMethode` est une méthode statique de la classe `nomClasse`
- pour appeler une méthode d'objet : on écrit `nomObjet.nomDeMethode(...)`

Par exemple, la classe `System` contient un attribut statique `out` de type `PrintStream`, classe contenant la méthode d'objet `println()`. Donc l'objet `System.out` appelle une méthode d'objet de nom `println()`.

Appel d'une méthode

- pour appeler une méthode statique : on écrit `nomClasse.nomMethode` si `nomMethode` est une méthode statique de la classe `nomClasse`
- pour appeler une méthode d'objet : on écrit `nomObjet.nomDeMethode(...)`

Par exemple, la classe `System` contient un attribut statique `out` de type `PrintStream`, classe contenant la méthode d'objet `println()`. Donc l'objet `System.out` appelle une méthode d'objet de nom `println()`.

Appel d'une méthode

- pour appeler une méthode statique : on écrit `nomClasse.nomMethode` si `nomMethode` est une méthode statique de la classe `nomClasse`
- pour appeler une méthode d'objet : on écrit `nomObjet.nomDeMethode(...)`

Par exemple, la classe `System` contient un attribut statique `out` de type `PrintStream`, classe contenant la méthode d'objet `println()`. Donc l'objet `System.out` appelle une méthode d'objet de nom `println()`.

Désigner l'objet courant dans une méthode d'instance

Une méthode d'instance est appelée par un objet de la classe dans laquelle est définie cette méthode.
Pour désigner cet objet qui appelle la méthode *dans le corps de la méthode* on utilise le mot clef `this`.
(voir la classe `Point` et la méthode `afficher`)

Surcharge

Une méthode de nom donné peut posséder plusieurs définitions une même classe, chacune de ces définitions se distinguant des autres au travers de la liste de ses paramètres.
Par contre le type du résultat n'intervient pas dans cette différenciation.
Par exemple, la classe Point pourrait avoir une autre méthode à 4 paramètres

```
public static boolean sontAlignes(Point A, Point B,
Point C, Point D){... }
```

Exemple

```
class EssaiThis {
// méthode d'instance
void faire(EssaiThis instance){
System.out.print(instance == this) ;
}
public static void main(String[] args) {
EssaiThis instance1 =new EssaiThis ;
EssaiThis instance2 =new EssaiThis ;
instance1.faire(instance2) ;
instance1.faire(instance1) ;
}}
```

Quel sera l'affichage à l'exécution de la méthode `main` ?

false true

Exemple

```
class EssaiThis {
// méthode d'instance
void faire(EssaiThis instance){
System.out.print(instance == this) ;
}
public static void main(String[] args) {
EssaiThis instance1 =new EssaiThis ;
EssaiThis instance2 =new EssaiThis ;
instance1.faire(instance2) ;
instance1.faire(instance1) ;
}}
```

Quel sera l'affichage à l'exécution de la méthode `main` ?

`false` `true`

Exemple

```
class EssaiThis {
// méthode d'instance
void faire(EssaiThis instance){
System.out.print(instance == this) ;
}
public static void main(String[] args) {
EssaiThis instance1 =new EssaiThis ;
EssaiThis instance2 =new EssaiThis ;
instance1.faire(instance2) ;
instance1.faire(instance1) ;
}}
```

Quel sera l'affichage à l'exécution de la méthode `main` ?

`false true`

Modificateur `final`

- `final` unAttribut : aucune modification de unAttribut après son initialisation
- `final` uneMéthode : aucune modification de uneMéthode après sa déclaration (donc pas de redéfinition dans une sous-classe)
- `static final ...` uneValeur = `...` : uneValeur est une constante.

Modificateur `final`

- `final` unAttribut : aucune modification de unAttribut après son initialisation
- `final` uneMéthode : aucune modification de uneMéthode après sa déclaration (donc pas de redéfinition dans une sous-classe)
- `static final ...` uneValeur = `...` : uneValeur est une constante.

Modificateur `final`

- `final` unAttribut : aucune modification de unAttribut après son initialisation
- `final` uneMéthode : aucune modification de uneMéthode après sa déclaration (donc pas de redéfinition dans une sous-classe)
- `static final` ... uneValeur = ... : uneValeur est une constante.

Java fournit 3 niveaux de protection pour les membres d'une classe. Chaque attribut et chaque méthode d'une classe peut être :

- visible depuis les instances de toutes les classes d'une application. En d'autres termes, son nom peut être utilisé dans l'écriture d'une méthode de ces classes. Les données peuvent modifiées par une méthode de la classe, d'une autre classe ou depuis la fonction `main`. Il est alors déclaré avec le modificateur `public`.
- visible uniquement depuis les instances de sa classe. En d'autres termes, son nom peut être utilisé **uniquement** dans l'écriture d'une méthode **de sa classe**. Il est alors déclaré avec le modificateur `private`.
- visible uniquement depuis les instances de sa classe et de ses sous-classes. Il est alors déclaré avec le modificateur `protected`.

Par défaut si les données sont déclarées sans type de protection, elles sont `public`.

bonnes habitudes

- les attributs ne doivent pas être visibles, ils ne pourront être lus ou modifiés que par l'intermédiaire de méthodes prenant en charge les vérifications et effets de bord éventuels.
- les méthodes "utilitaires" ne doivent pas être visibles, seules les fonctionnalités de l'objet, destinées à être utilisées par d'autres objets soient visibles.

C'est la notion d'*encapsulation* : les données doivent pouvoir être contrôlées ainsi que les comportement de l'objet. On vient de voir la protection des données. Maintenant on va voir le contrôle de l'accès puis les constructeurs.

Méthodes d'accès aux données

Lorsque les données sont totalement protégées `private`, elles ne sont plus accessibles depuis une autre classe ou depuis la fonction `main` d'un autre programme.
Il faut donc créer *dans la classe* des méthodes d'accès aux données de la classe.
Il y a deux types d'accès à envisager :

- en consultation ou lecture, on parle d'accesseur en consultation,
- en modification ou écriture, on parle d'accesseur en modification.

Par convention les méthodes d'accès en lecture doivent commencer par `get` et les méthodes d'accès en écriture doivent commencer par `set`.
(voir la classe `Point`)

Constructeurs

Un constructeur est une méthode particulière qui sert à initialiser un objet.
Si on omet d'écrire explicitement un constructeur dans une classe alors Java crée un *constructeur par défaut* qui a les caractéristiques suivantes :

- il a le nom de la classe
- son en-tête ne contient aucun type de retour ni `void`
- il n'a pas de paramètre
- il a un corps vide
- il est invoqué avec new

Java permet la définition de plusieurs constructeurs dans une même classe qui auront tous le même nom que le constructeur par défaut : on dit alors que le constructeur est *surchargé* (lorsqu'il y en a au moins deux).

Constructeurs

ATTENTION : dès qu'un constructeur est explicitement définie dans une classe, le constructeur par défaut n'existe plus.
Donc si on veut avoir une version du constructeur par défaut parmi d'autres constructeur, il faut le mettre explicitement. ((voir la classe `Point`)

Disparition d'un objet

Un objet disparaît lorsque plus aucune référence sur lui n'existe. L'espace qu'il occupe en mémoire peut alors être récupéré. Java possède un *ramasse-miettes* (*garbage collector*) qui se charge de cette récupération sans intervention nécessaire du programmeur.

La classe `String`

La classe `String` est une classe prédéfinie de Java.
Elle comporte de nombreuses méthodes permettant la manipulation de chaînes de caractères.
C'est une classe *finale* c'est-à-dire que ses méthodes sont elles aussi finales et ne peuvent donc pas être surchargées (on ne peut pas utiliser des méthodes de même nom).
Les objets de la classe `String` sont *immuables*, c'est-à-dire qu'ils ne peuvent pas être modifiés, mais, à chaque modification est créé un nouvel objet.
Les méthodes de la classe `String` sont nombreuses (voir l'API).

Quelques méthodes

- `int length()` $\rightsquigarrow$ renvoie la longueur de la chaîne objet (soit le nombre de caractères espace compris),
- `char charAt(int index)` $\rightsquigarrow$ renvoie le caractère en position index sachant que le premier caractère de la chaîne est en position 0,
- `String toLowerCase()` $\rightsquigarrow$ renvoie une chaîne composée des mêmes caractères que l'objet en minuscules,
- `String toUpperCase()` $\rightsquigarrow$ renvoie une chaîne composée des mêmes caractères que l'objet en majuscules.

`"abc".charAt(1)` retourne le caractère `b`
`"X2001".toLowerCase()` retourne la chaîne `x2001`.

méthodes de comparaison

- `boolean equals(String s)` $\rightsquigarrow$ teste l'égalité de l'objet `String` à `s`
- `int compareTo(String s)` $\rightsquigarrow$ compare l'objet à `s` selon l'ordre alphabétique ; elle retourne un entier **négatif** si l'objet est **avant s**, **0** si l'objet est **égal à s** et un entier **positif** si l'objet est **après s**.
- `boolean startsWith(String prefix)` $\rightsquigarrow$ teste si l'objet commence avec la chaîne prefix
- `boolean endsWith(String suffix)` $\rightsquigarrow$ teste si l'objet termine avec la chaîne suffix
- `int indexOf(String facteur)` $\rightsquigarrow$ retourne -1 si facteur n'est pas un facteur de l'objet et retourne le plus petit indice du début de facteur dans l'objet sinon.
- `int lastIndexOf(String facteur)` $\rightsquigarrow$ retourne -1 si facteur n'est pas un facteur de l'objet et retourne le plus grand indice du début de facteur dans l'objet sinon.

Exemples

`"chocolat".compareTo("chou à la crème")` renvoie un nombre négatif

`"A200002".indexOf("00")` renvoie 2

`"A200002".lastIndexOf("00")` renvoie 4.

méthodes de modification

- `String substring(int debut, int fin)` $\rightsquigarrow$ renvoie la sous-chaîne de l'objet depuis l'indice debut jusqu'à l'indice fin-1.
- `String concat(String s)` $\rightsquigarrow$ crée une nouvelle chaîne composée de l'objet suivi de s.
- `String replace(char x, char y)` $\rightsquigarrow$ crée une nouvelle chaîne en remplaçant dans l'objet toutes les occurrences du caractère x par le caractère y.

`"chocolat".substring(2,5)` est la chaîne "oco".
`"j'aime ".concat("le chocolat")` est la chaîne "j'aime le chocolat".
`"chocolat".replace('o','u')` est la chaîne "chuculat".

Remarque : le terme modification n'est pas très appropriée puisqu'il y a en fait création d'une nouvelle chaîne.

Remarque

Les arguments de la méthode `System.out.println` sont de type `String`.
Pourtant on a jusqu'à présent mélanger des variables de type primitif et des chaînes de caractères en les séparant par + : en fait, avec cette méthode tout nombre est converti en une chaîne de caractère composée des chiffres qui le composent et de même tout booléen est transformé en la chaîne `true` ou `false` selon sa valeur.

La classe `StringBuffer`

Si l'on veut modifier l'intérieur d'une chaîne sans en créer une nouvelle il faut utiliser la classe `StringBuffer` : `StringBuffer` est une classe du paquetage java.lang comme la classe `String`.
Elle possède aussi la méthode `char charAt(int index)` qui retourne le caractère d'indice index.
Pour modifier une chaîne on peut utiliser la méthode `void setCharAt(int index, char nouveau)` qui remplace le caractère d'indice index de l'objet par le caractère nouveau.

lien avec String

Un constructeur de la classe `StringBuffer` accepte en paramètre un objet de la classe `String`.

```
String s="hello";
StringBuffer sbf = new StringBuffer(s);
```

Inversement la méthode d'instance `toString` invoqué par un objet de type `StringBuffer` renvoie un objet de type `String` référençant la même chaîne que l'objet de type `StringBuffer`.

```
StringBuffer sbf = new StringBuffer("Where is it?");
String s = sbf.toString();
```

Classe interne

Une classe interne est une classe définie à l'intérieur d'une autre classe. Une classe interne a un accès à tous les attributs de ces classes englobantes.
Depuis l'extérieur on pourra faire référence à une méthode d'une classe interne en préfixant son identificateur de la façon suivante :
ClasseExterne.ClasseInterne.uneMéthodeInterne();

Chapitre V – Héritage

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Chapitre V – Héritage

- I. Principe
- II. Un exemple
- III. Constructeurs dans une sous-classe
- IV. Polymorphisme
- V. Conversion de classe ou Transtypage
- VI. Redéfinir des méthodes
- VII. Masquage d'attributs
- VIII. La classe `Object`

Principe

- toute classe hérite d'une et d'une seule classe appelée sa *superclasse* hormis la classe `java.lang.Object`
- une classe qui n'indique pas sa superclasse hérite automatiquement de la classe `Object`.

Syntaxe

```
class B extends A { ....
```

On dit que B *hérite* de A ou que B est une *sous-classe* de A ou que A est la *superclasse* de B ou que B *étend* A.
B devient un ≪ sous-ensemble ≫ de A.
B peut alors utiliser tous les attributs et méthodes de sa superclasse A.
B peut aussi définir des attributs supplémentaires et ses propres méthodes qui distingueront les objets B des objets A.
Mais attention, si un attribut est déclaré `private` dans la superclasse A alors cet attribut devient un attribut de la classe B mais il n'est pas directement consultable ou modifiable dans la classe B. Il faut alors utiliser des accesseurs de la superclasse A pour manipuler ces attributs ou bien alors modifier la protection avec `protected` dans la superclasse A.

Schéma d'héritage

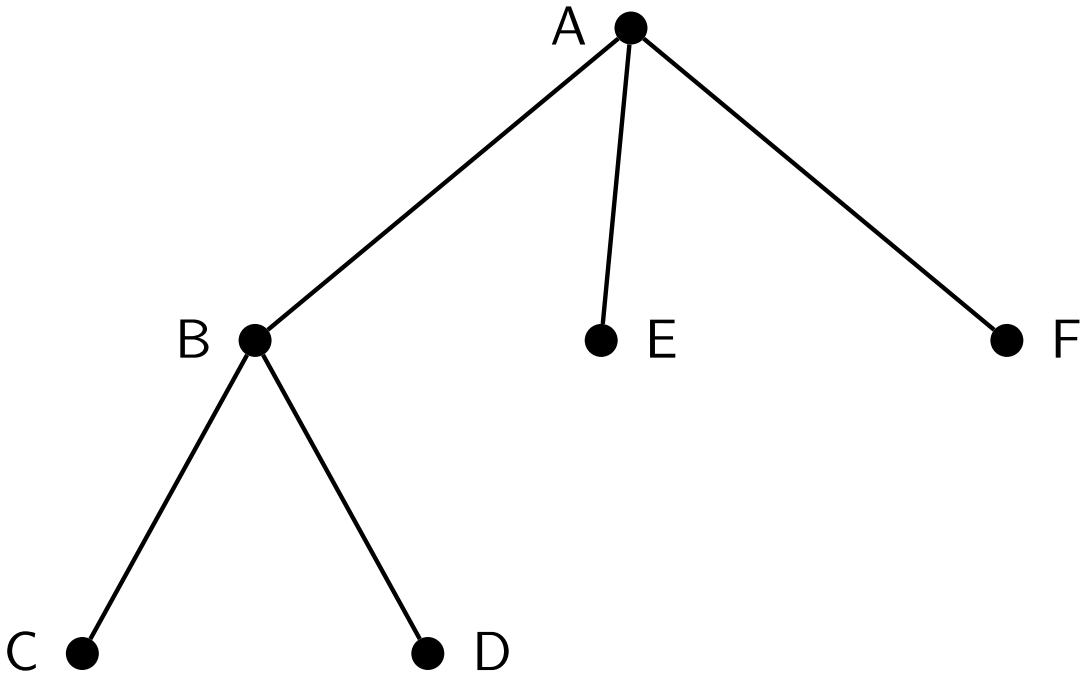
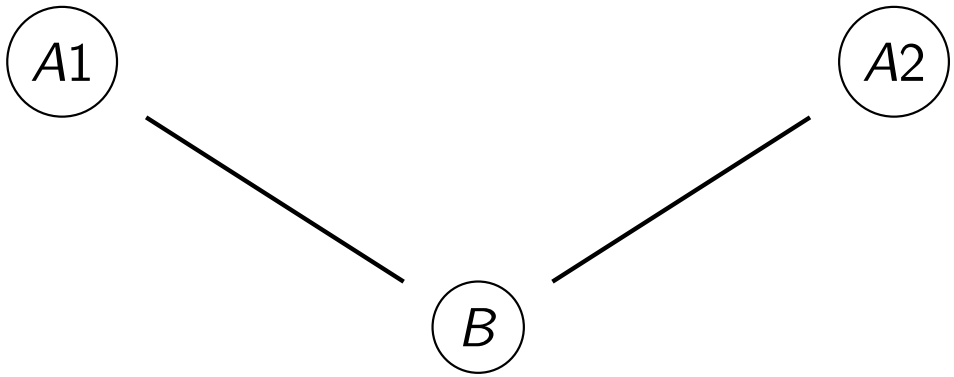


Schéma impossible



Interdiction d'héritage

```
final class ClasseFinale { ...
```

On peut interdire à une classe d'être étendue en la déclarant `final`

Un exemple

```
public class EtudiantStand{
// attributs
private String nom;
private String prenom;
private int numeroCarte;
//Constructeurs
public EtudiantStand(String nom, String prenom, int numeroCarte){
this.nom=nom;this.prenom= prenom; this.numeroCarte=numeroCarte;
}
public EtudiantStand(){}
//méthodes d'accès
public void setNom(String sonNom){
this.nom=sonNom;}
public String getNom(){
return this.nom;}
public void setPrenom(String sonPrenom){
this.nom=sonPrenom;}
public String getPrenom(){
return this.prenom;}
```

Un exemple

```
public void setNumeroCarte(int x){
this.numeroCarte=x;}
public int getNumeroCarte(){
return this.numeroCarte;}
public void afficheEtudiant(){
System.out.println("nom : " + this.getNom()) ;
System.out.println("prenom : " + this.getPrenom()) ;
System.out.println("numero de carte d etudiant : "
+ this.getNumeroCarte()) ;}}
```

(suite)

```
public class Etudiant extends EtudiantStandard {
String specialite;
int anneeDeFormation ;
// à compléter}
```

Constructeurs dans une sous-classe

Règle : TOUT CONSTRUCTEUR D'UNE CLASSE AUTRE QUE LA CLASSE `Object` FAIT APPEL SOIT À UN CONSTRUCTEUR DE SA SUPERCLASSE SOIT À UN AUTRE CONSTRUCTEUR DE SA CLASSE.
Cet appel est nécessairement la première instruction de constructeur.
De plus un appel à

- un constructeur de la superclasse se fait à l'aide du mot `super`,
- un constructeur de la classe se fait à l'aide du mot `this`.

Constructeurs dans une sous-classe

Règle : TOUT CONSTRUCTEUR D'UNE CLASSE AUTRE QUE LA CLASSE `Object` FAIT APPEL SOIT À UN CONSTRUCTEUR DE SA SUPERCLASSE SOIT À UN AUTRE CONSTRUCTEUR DE SA CLASSE. Cet appel est nécessairement la première instruction de constructeur. De plus un appel à

- un constructeur de la superclasse se fait à l'aide du mot `super`,
- un constructeur de la classe se fait à l'aide du mot `this`.

Constructeurs dans une sous-classe

Règle : TOUT CONSTRUCTEUR D'UNE CLASSE AUTRE QUE LA CLASSE `Object` FAIT APPEL SOIT À UN CONSTRUCTEUR DE SA SUPERCLASSE SOIT À UN AUTRE CONSTRUCTEUR DE SA CLASSE. Cet appel est nécessairement la première instruction de constructeur. De plus un appel à

- un constructeur de la superclasse se fait à l'aide du mot `super`,
- un constructeur de la classe se fait à l'aide du mot `this`.

exemple (suite)

```
Etudiant (String nom, String prenom, int numeroCarte,
String specialite, int anneeDeFormation){
super(nom, prenom, numeroCarte);
this.specialite = specialite;
this.anneeDeFormation = anneeDeFormation}
....
Etudiant toto = new Etudiant("Smith", "John",
20113245, "Miage", 3);
....
```

si aucun constructeur de la sous-classe n'est défini, le compilateur ajoute le constructeur suivant :

```
B(){
  super();
}
```

Si un constructeur de la sous-classe oublie en première instruction l'appel à un constructeur de la super-classe, le compilateur ajoute l'instruction **super**() ; en première ligne du constructeur de la sous-classe.

Donc attention il faut que dans la super-classe il existe un tel constructeur sans paramètre.

Les constructeurs sont chaînées et les appels aux constructeurs remontent de super-classe en super-classe jusqu'à la classe `Object`.

Dans l'ordre, le constructeur de la classe parent s'exécute avant le constructeur de la classe enfant.

si aucun constructeur de la sous-classe n'est défini, le compilateur ajoute le constructeur suivant :

```
B(){
  super();
}
```

Si un constructeur de la sous-classe oublie en première instruction l'appel à un constructeur de la super-classe, le compilateur ajoute l'instruction **super**(); en première ligne du constructeur de la sous-classe.

Donc attention il faut que dans la super-classe il existe un tel constructeur sans paramètre.

Les constructeurs sont chaînées et les appels aux constructeurs remontent de super-classe en super-classe jusqu'à la classe `Object`.

Dans l'ordre, le constructeur de la classe parent s'exécute avant le constructeur de la classe enfant.

si aucun constructeur de la sous-classe n'est défini, le compilateur ajoute le constructeur suivant :

```
B(){
  super();
}
```

Si un constructeur de la sous-classe oublie en première instruction l'appel à un constructeur de la super-classe, le compilateur ajoute l'instruction **super**() ; en première ligne du constructeur de la sous-classe.

Donc attention il faut que dans la super-classe il existe un tel constructeur sans paramètre.

Les constructeurs sont chaînées et les appels aux constructeurs remontent de super-classe en super-classe jusqu'à la classe `Object`.

Dans l'ordre, le constructeur de la classe parent s'exécute avant le constructeur de la classe enfant.

Polymorphisme

Tout objet a un type **déclaré** qui sera le type de sa référence.
Tout objet a un type **réel**, celui de son constructeur. Mais il peut appartenir à plusieurs classes.
Si A est la classe parent des classes B et C, alors une variable de type A peut être instanciée par un objet de type B ou C. On dit que le type A est *polymorphe*.
L'opérateur `instanceof` permet de tester l'appartenance d'un objet à une classe.

```
A a;// a est de type déclaré A
a = new B();// a est de type réel B
System.out.println("Objet de la classe A : " +
(a instanceof A));//true
System.out.println("Objet de la classe B : " +
 (a instanceof B));//true
System.out.println("Objet de la classe C : " +
(a instanceof C));//false
```

Conversion de classe

Si B hérite de A, un objet de type B est toujours un objet de type A mais un objet de type A n'est pas nécessairement un objet de type B.

```
A a;
B b;
a=b; // oui (upcasting implicite)
b=a; // ne compile pas
```

On peut convertir explicetement un objet B en objet A de la façon suivante :

```
a = (A) b;
```

Dans l'autre sens, si on est sûr que l'objet A peut être considéré comme un objet B, on peut faire une conversion de B vers A (downcasting).

On en verra l'utilité dans la méthode `clone()`.

Si une conversion est impossible une exception `CastClassException` est levée à l'exécution.

Redéfinir des méthodes

Règles :

- on ne peut pas redéfinir une méthode dans une sous-classe en modifiant le type de retour, en diminuant sa visibilité avec un modificateur de protection plus fort ou en la définissant de méthode d'instance à méthode statique ou inversement.
- une méthode redéfinie possède le même nom, les mêmes paramètres et le même type de retour

En cas de redéfinition d'une méthode, à l'exécution

- la méthode d'instance s'appliquera selon le **type réel** de l'objet qui l'appelle (*liaison tardive*)
- la méthode statique s'appliquera selon son **préfixe-classe**

exemple(suite)

```
// class Etudiant suite
public void afficheEtudiant(){// méthode redéfinie
System.out.println("nom : " + this.getNom()) ;
System.out.println("prenom : " + this.getPrenom()) ;
System.out.println("numero de carte d etudiant : "
+ this.getNumeroCarte()) ;
System.out.println("specialite : " + this.getSpecialite()) ;
System.out.println("annee : " + this.getAnneeDeFormation()) ;
}
```

Surcharge

Attention : ne pas confondre surcharge et redéfinition.
Une méthode est surchargée si elle a le même nom mais un type de retour et/ou des paramètres différents.
L'appel a une méthode d'instance surchargée se fait selon le type déclaré de l'objet appelant.

Masquage d'attributs

Une sous-classe peut définir un attribut de même nom que sa classe parent : il y a masquage d'attribut.
Une variable d'instance masquée utilisée dans une classe est la variable d'instance connue statiquement selon la **classe** de l'objet.
A EVITER

Object

Toute classe hérite de la classe `Object`.

- Constructeur : `Object();`
- Méthodes :
  - `public String toString();`
  - `public boolean equals (Object obj);`
  - `protected Object clone();`
  - ...

toString

La méthode d'instance `toString` renvoie le nom de la classe de l'objet suivi de la référence de cet objet. Cette méthode est utilisée par la méthode `System.out.print`.
On peut la redéfinir.

toString

```
public class Point{
float abs, ord;
Point(float x, float y){ this.abs =x;this.ord = y;}
public String toString(){
return("abscisse:"+this.abs+",ordonnée:"+this.ord);}}
...
Point monPoint = new(2,1);
System.out.println(monPoint) ;
//abscisse : 2.0, ordonnée :1.0
```

Une sous-classe peut redéfinir la méthode `toString` en utilisant la redéfinition de sa superclasse en appelant `super.toString`.

equals

== compare

- les valeurs pour les types primitifs
- les références pour les objets ou les tableaux

Si on veut une comparaison en « profondeur » , il est nécessaire de redéfinir ==.

```
public boolean equals(Point autrePoint){
return(this.abs==autrePoint.abs && this.ord==autrePoint.ord);}
```

equals

== compare

- les valeurs pour les types primitifs
- les références pour les objets ou les tableaux

Si on veut une comparaison en « profondeur » , il est nécessaire de redéfinir ==.

```
public boolean equals(Point autrePoint){
return(this.abs==autrePoint.abs && this.ord==autrePoint.ord);}
```

Chapitre VI – Documenter un projet

Chapitre VI – Documenter un projet

Généralités

On peut annoter son programme pour obtenir un fichier html de commentaires.

- commencer par $/**$, terminer par $*/$ (entre les deux commencer chaque ligne par $*$).
- @auteur : il peut y avoir plusieurs auteurs
- @see : pour créer un lien vers un autre document
- @param : pour indiquer les paramètres d'une méthode
- @exception :pour indiquer quelle exception est levée
- @return : pour indiquer la valeur de retour d'une méthode
- @version : pour donner le numéro de la version du code
- @since : pour donner le numéro de la version initiale
- @deprecated : pour indiquer qu'une méthode ne devrait plus être utilisée, (cela crée un warning à la compilation)

commande

Après compilation de `MonProgramme.java`, la commande
`javadoc MonProgramme.java`
engendre un fichier `MonProgramme.html`.
Options de la commande `javadoc` :

- -author : Indique que les commentaires tagés par `@author` devront être utilisées pour générer la documentation (par défaut, ces informations ne sont pas utilisées).
- -d repertoire : Permet de préciser le repertoire où javadoc placera les fichiers HTML générés (par défaut, répertoire courant).
- -public : Ne documente que les membres (méthodes, constructeurs, attributs) publics.
- -private : Documente tous les membres (méthodes, constructeurs, attributs), quelle que soit leur visibilité (par défaut membres publics et protected)

Un exemple

```
import java.util.*;
/** Le premier exemple de programme Java.
* Affiche une chaîne de caractères et la date du jour.
* @author moi
*@author http://www.lacl.u-pec/moi/
*@version 0.0 */
public class BonjourDate {
/** Unique point d'entrée de la classe et de l'application
*@param args tableau de paramètres sous forme de chaînes de caractères
*@return Pas de valeur de retour
*@exception exceptions Pas d'exceptions émises */
public static void main(String[] args) {
System.out.println("Bonjour, aujourd'hui: ");
System.out.println(new Date());}}
```

suite

- index.html
- allclasses-frame.html
- allclasses-noframe.html
- constant-values.html
- deprecated-list.html
- BonjourDate.html
- help-doc.html
- index-all.html
- overview-tree.html
- package-frame.html
- package-summary.html
- package-tree.html
- package-list
- stylesheet.css

Chapitre VII – Exceptions

Chapitre VII – Exceptions

- I. Exceptions prédéfinies
- II. Définir ses propres exceptions
- III. Lancer une exception
- IV. Attraper une exception
- V. Propagation d'une exception

Chapitre VII – Exceptions

- I. Exceptions prédéfinies
- II. Définir ses propres exceptions
- III. Lancer une exception
- IV. Attraper une exception
- V. Propagation d'une exception

Chapitre VII – Exceptions

- I. Exceptions prédéfinies
- II. Définir ses propres exceptions
- III. Lancer une exception
- IV. Attraper une exception
- V. Propagation d'une exception

Chapitre VII – Exceptions

- I. Exceptions prédéfinies
- II. Définir ses propres exceptions
- III. Lancer une exception
- IV. Attraper une exception
- V. Propagation d'une exception

Qu'est-ce qu'une exception

En Java, une exception :

- est un objet, instance d'une classe d'exception
- est créee par un erreur
- contient des informations sur cette erreur
- provoque la sortie d'une méthode
- se propage de méthodes en méthodes
- peut être capturée et traitée
- peut provoquer l'arrêt du programme si aucun traitement.

Qu'est-ce qu'une exception

En Java, une exception :

- est un objet, instance d'une classe d'exception
- est créee par un erreur
- contient des informations sur cette erreur
- provoque la sortie d'une méthode
- se propage de méthodes en méthodes
- peut être capturée et traitée
- peut provoquer l'arrêt du programme si aucun traitement.

Qu'est-ce qu'une exception

En Java, une exception :

- est un objet, instance d'une classe d'exception
- est créee par un erreur
- contient des informations sur cette erreur
- provoque la sortie d'une méthode
- se propage de méthodes en méthodes
- peut être capturée et traitée
- peut provoquer l'arrêt du programme si aucun traitement.

Qu'est-ce qu'une exception

En Java, une exception :

- est un objet, instance d'une classe d'exception
- est créee par un erreur
- contient des informations sur cette erreur
- provoque la sortie d'une méthode
- se propage de méthodes en méthodes
- peut être capturée et traitée
- peut provoquer l'arrêt du programme si aucun traitement.

Qu'est-ce qu'une exception

En Java, une exception :

- est un objet, instance d'une classe d'exception
- est créee par un erreur
- contient des informations sur cette erreur
- provoque la sortie d'une méthode
- se propage de méthodes en méthodes
- peut être capturée et traitée
- peut provoquer l'arrêt du programme si aucun traitement.

Qu'est-ce qu'une exception

En Java, une exception :

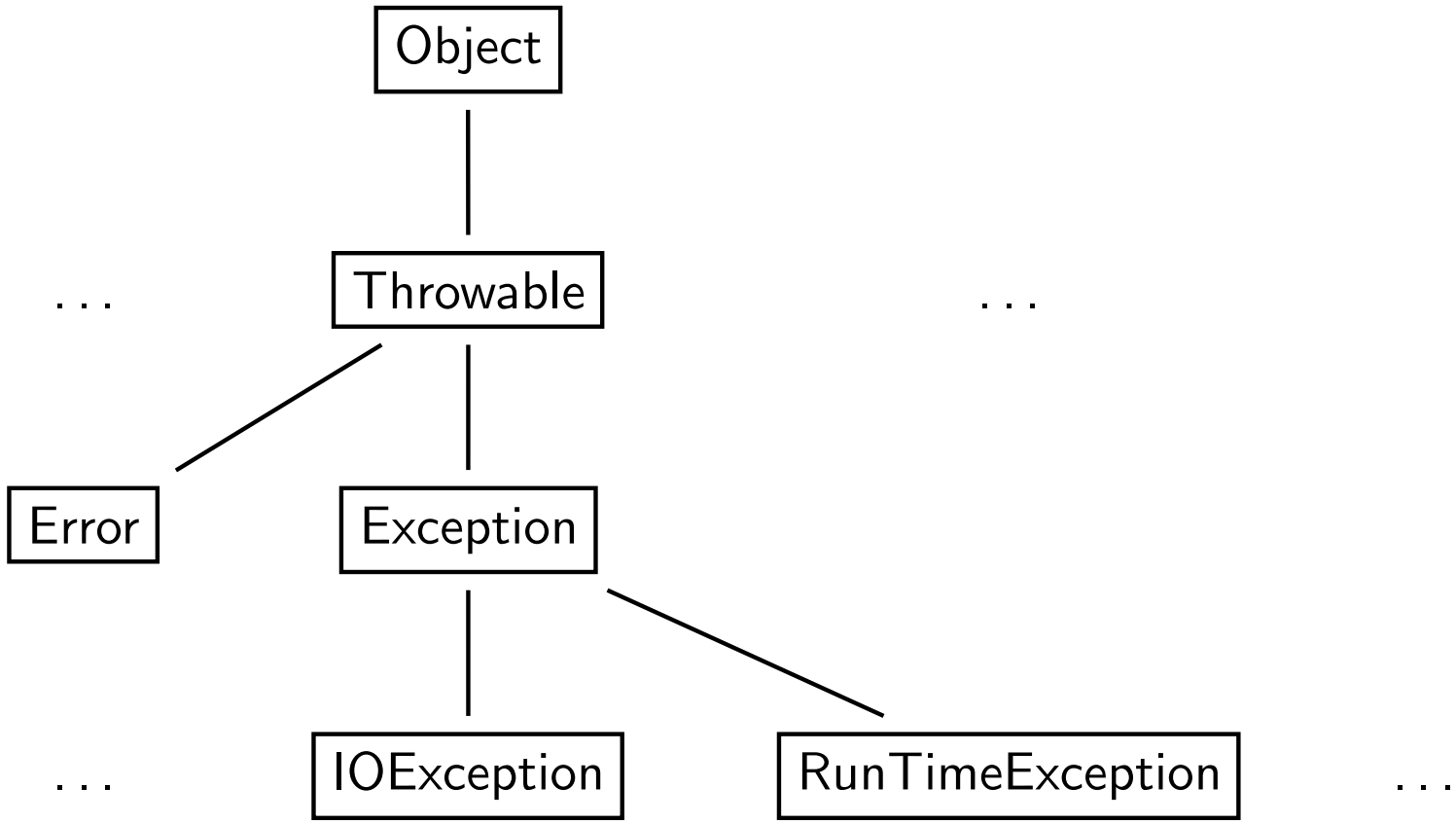
- est un objet, instance d'une classe d'exception
- est créee par un erreur
- contient des informations sur cette erreur
- provoque la sortie d'une méthode
- se propage de méthodes en méthodes
- peut être capturée et traitée
- peut provoquer l'arrêt du programme si aucun traitement.

Qu'est-ce qu'une exception

En Java, une exception :

- est un objet, instance d'une classe d'exception
- est créee par un erreur
- contient des informations sur cette erreur
- provoque la sortie d'une méthode
- se propage de méthodes en méthodes
- peut être capturée et traitée
- peut provoquer l'arrêt du programme si aucun traitement.

Exceptions prédéfinies



Exceptions prédéfinies

La classe Error contient les erreurs graves : StackOverflowError, ...
Les erreurs les plus souvent lancées font parties de la classe RunTimeException :

- ArithmeticException
- IllegalArgumentException
- ClassCastException
- IndexOutOfBoundException
- NullPointerException
- ...

On y trouve des erreurs prédéfinies :

- Division par zéro pour les entiers : ArithmeticException
- Référence nulle : NullPointerException
- Tentative de forçage de type illégale : ClassCastException
- Tentative de création d'un tableau de taille négative : NegativeArraySizeException
- Dépassement de limite d'un tableau : ArrayIndexOutOfBoundsException

exemple

```
public class Addition{
public static void main (String[] args){
int somme = 0 ;
for (int ix = 0 ; ix < args.length; ix++) {
somme += Integer.parseInt(args[ix]) ;}
System.out.println("somme=" + somme);} // main
}
```

```
 java Addition 1 2 trois 4
Exception in thread "main" java.lang.NumberFormatException:
For input String "trois"
at
java.lang.NumberFormatException.forInputString(NumberFormatExcept
at java.lang.Integer.parseInt(Integer.java:481)
at java.lang.Integer.parseInt(Integer.java:514)
at Addition.main(Addition.java:7)
```

Définir ses propres exceptions

Règles : Pour définir ses propres exceptions on crée une sous-classe de la classe Exception.

```
class MonException extends Exception { ...}
```

Lancer une exception

Règles :

- créer une instance de la classe d'exception que l'on veut lancer
- lancer une exception avec le mot clé throw

```
throw new MonException() ;
```

Si cette instruction est exécutée alors l'exécution normale du programme s'arrête :
▷ soit l'exception lancée est attrapée dans une des méthodes appelant le bloc en cours
▷ soit on sort du programme

Règle : Java impose de déclarer qu'une méthode peut lancer une exception sans l'attraper sauf pour les exceptions des classes Error et RuntimeException avec la clause throws.

```
 public class MonException extends Exception}{
 MonException(){System.out.println("ceci est une erreur");}
 }
public class EssaiException {
public static void main throws MonException  (String[] args){
System.out.println("message 1");
if (true) throw new MonException() ;
System.out.println("message 2");
System.out.println("message 3");
```

exécution :

```
message 1
ceci est une erreur
Exception in thread "main" EssaiException at
EssaiException.main(EssaiException.java: 4)
```

Attraper une exception

Règle : Pour attraper une exception on utilise deux blocs consécutifs

- try {...} : contient l'instruction throw
- catch(MonException e) {...} : contient les instructions à exécuter si le bloc try a lancé une instance de MonException

```
try {
suiteInstruc1;
...throw new MonException()...;
suiteInstruc2; }
catch(MonException e) {suiteInstr3;}
suiteInstruc4;
```

throw n'est pas exécuté : suiteInstruc1, suiteInstruc2, suiteInstruc4 sont successivement exécutées

throw est exécuté : suiteInstruc1, suiteInstruc3, suiteInstruc4 sont successivement exécutées

une exception d'un autre type est lancée : message d'erreur ou traitement par un bloc appelant

Exemple

```
public class Addition{
public static void main (String[] args){
int somme = 0 ;
for (int ix = 0 ; ix < args.length; ix++) {
try{int k = Integer.parseInt(args[ix]);
     somme += k;}
catch(Exception e){} }
System.out.println("somme=" + somme);} // main
}
```

exécution :

```
java Addition 1 2 trois 4
somme = 7
```

Attraper plusieurs exceptions

On peut enchaîner les blocs `catch` si le bloc `try` est susceptible de lancer plusieurs types d'exception.
On peut imbriquer un bloc `try catch` dans un bloc `try`

Bloc finally

Règle : un bloc try est nécessairement suivi soit d'un bloc finally soit d'un ou plusieurs blocs catch suivis éventuellement d'un bloc finally.
Le bloc `finally` contient des instructions qui seront exécutées à la sortie du bloc `try` qui le précède immédiatement ou bien après l'exécution du bloc `catch` intercalé entre les blocs `try` et `finally`.
On a donc soit

- **try** + **finally**
- **try** + **catch**
- **try** + **catch** + **finally**

Quelque soit la façon dont on est sorti du bloc `try` le bloc `finally` qui suit est exécuté juste après cette sortie.

Propagation d'une exception

Si une méthode mB est appelée par une méthode mA et si mB lance une exception sans l'attraper alors la méthode mA peut attraper l'exception lancée par mB : il y a propagation de l'exception.
Le code doit suivre le schéma de l'exemple suivant :

```
class ExceptPropag {
static void methodeNiveauC () throws MonException {
try {
if (true) throw new MonException() ;
System.out.println(" ? ? ? C") ;
    }
finally {
System.out.println("niveau C") ;
}}
static void methodeNiveauB ()  throws MonException {
try {
methodeNiveauC() ;
System.out.println( " ? ? ? B ") ;
    }
finally {
System.out.println("niveau B") ;
}}
static void methodeNiveauA () {
try {
methodeNiveauB() ;
 System.out.println( " ? ? ? A ") ;
   }
catch(MonException e){
System.out.println( "exception attrape au niveau A") ;
e.printStackTrace() ;
}
System.out.println( "on reprend comme si de rien était") ;} }
```

Exécution de `methodeNiveauA()` dans une classe `EssaiExceptPropag` :

```
ceci est une erreur
niveau C
niveau B
exception attrapée au niveau A
MonException
        : at ExceptPropag.methodeNiveauC (ExceptPropag. java :
4)
        : at ExceptPropag.methodeNiveauB (ExceptPropag. java :
13)
        : at ExceptPropag.methodeNiveauA (ExceptPropag. java :
22)
        : at EssaiExceptPropag.main (EssaiExceptPropag.java : 3 )
on reprend comme si de rien n'etait
```

(voir méthodes de la classe Throwable : printStackTrace() ...)

Exécution de `methodeNiveauA()` dans une classe `EssaiExceptPropag` :

ceci est une erreur
niveau C
niveau B
exception attrapée au niveau A
MonException
: at ExceptPropag.methodeNiveauC (ExceptPropag. java : 4)
: at ExceptPropag.methodeNiveauB (ExceptPropag. java : 13)
: at ExceptPropag.methodeNiveauA (ExceptPropag. java : 22)
: at EssaiExceptPropag.main (EssaiExceptPropag.java : 3)
on reprend comme si de rien n'etait

(voir méthodes de la classe Throwable : printStackTrace() ...)

Exécution de `methodeNiveauA()` dans une classe `EssaiExceptPropag` :

ceci est une erreur
niveau C
niveau B
exception attrapée au niveau A
MonException
: at ExceptPropag.methodeNiveauC (ExceptPropag. java : 4)
: at ExceptPropag.methodeNiveauB (ExceptPropag. java : 13)
: at ExceptPropag.methodeNiveauA (ExceptPropag. java : 22)
: at EssaiExceptPropag.main (EssaiExceptPropag.java : 3)
on reprend comme si de rien n'etait

(voir méthodes de la classe Throwable : printStackTrace() ...)

Exécution de `methodeNiveauA()` dans une classe `EssaiExceptPropag` :

ceci est une erreur
niveau C
niveau B
exception attrapée au niveau A
MonException
: at ExceptPropag.methodeNiveauC (ExceptPropag. java : 4)
: at ExceptPropag.methodeNiveauB (ExceptPropag. java : 13)
: at ExceptPropag.methodeNiveauA (ExceptPropag. java : 22)
: at EssaiExceptPropag.main (EssaiExceptPropag.java : 3)
on reprend comme si de rien n'etait

(voir méthodes de la classe Throwable : printStackTrace() ...)

Exécution de `methodeNiveauA()` dans une classe `EssaiExceptPropag` :

ceci est une erreur
niveau C
niveau B
exception attrapée au niveau A
MonException
: at ExceptPropag.methodeNiveauC (ExceptPropag. java : 4)
: at ExceptPropag.methodeNiveauB (ExceptPropag. java : 13)
: at ExceptPropag.methodeNiveauA (ExceptPropag. java : 22)
: at EssaiExceptPropag.main (EssaiExceptPropag.java : 3)
on reprend comme si de rien n'etait

(voir méthodes de la classe Throwable : printStackTrace() ...)

Exécution de `methodeNiveauA()` dans une classe `EssaiExceptPropag` :

ceci est une erreur
niveau C
niveau B
exception attrapée au niveau A
MonException
: at ExceptPropag.methodeNiveauC (ExceptPropag. java : 4)
: at ExceptPropag.methodeNiveauB (ExceptPropag. java : 13)
: at ExceptPropag.methodeNiveauA (ExceptPropag. java : 22)
: at EssaiExceptPropag.main (EssaiExceptPropag.java : 3)
on reprend comme si de rien n'etait

(voir méthodes de la classe Throwable : printStackTrace() ...)

Exécution de `methodeNiveauA()` dans une classe `EssaiExceptPropag` :

ceci est une erreur
niveau C
niveau B
exception attrapée au niveau A
MonException
: at ExceptPropag.methodeNiveauC (ExceptPropag. java : 4)
: at ExceptPropag.methodeNiveauB (ExceptPropag. java : 13)
: at ExceptPropag.methodeNiveauA (ExceptPropag. java : 22)
: at EssaiExceptPropag.main (EssaiExceptPropag.java : 3)
on reprend comme si de rien n'etait

(voir méthodes de la classe Throwable : printStackTrace() ...)

Exécution de `methodeNiveauA()` dans une classe `EssaiExceptPropag` :

```
ceci est une erreur
niveau C
niveau B
exception attrapée au niveau A
MonException
        : at ExceptPropag.methodeNiveauC (ExceptPropag. java :
4)
        : at ExceptPropag.methodeNiveauB (ExceptPropag. java :
13)
        : at ExceptPropag.methodeNiveauA (ExceptPropag. java :
22)
        : at EssaiExceptPropag.main (EssaiExceptPropag.java : 3 )
on reprend comme si de rien n'etait
```

(voir méthodes de la classe Throwable : printStackTrace() ...)

Chapitre VIII – Types énumérés

Définition

Un *type énuméré* est un type qui permet de définir une variable comme élément d'un ensemble fini de valeurs constantes.
Les classes qui implémentent ces types diffèrent des autres classes dans leur déclaration : leur en-tête doit commencer par `enum` au lieu de `class`.
De plus, les identificateurs des champs d'un type énuméré sont écrits en majuscule car ce sont des constantes.
Chaque élément d'une énumération est un objet à part entière

Exemple

```
public enum ArcEnCiel{
ROUGE, ORANGE, JAUNE, VERT, BLEU, INDIGO, VIOLET
}

public class EssaiArcEnCiel{
public static void main(String[] arg) {
ArcEnCiel ace =  ArcEnCiel.valueOf(arg[0]);
if(ace.toString() == "ROUGE") System.out.println("chaud");
else if(ace==ArcEnCiel.BLEU)) System.out.println("froid");
System.out.println(ace.name());
System.out.println(ace.ordinal());
}
// toString() renvoie la chaîne de caractères identifiant la constante dans l'énumération
//ArcEnCiel.BLEU est une constante (statique) de type ArcenCiel
}
```

A l'exécution de `java EssaiArcEnCiel BLEU` on obtient `froid bleu 4`

java.lang.Enum

Règle : toute classe enum hérite automatiquement de la classe java.lang.Enum.
Par conséquent, tout type énuméré ne peut étendre une autre classe.
Toute classe enum hérite en particulier des méthodes suivantes :

- `toString()` : renvoie le nom de l'objet défini dans l'énumération
- `equals()` : teste l'égalité
- `valueOf(String s)` : méthode statique finale qui renvoie un objet du type énuméré dont le nom correspond à la chaîne s
- `values()` : méthode statique finale qui renvoie un tableau des valeurs énumérées dans l'ordre d'énumération
- `ordinal()` : méthode finale qui renvoie le numéro d'ordre dans l'énumération en commençant par 0
- `name()` : méthode finale qui renvoie la chaîne de caractères du nom de l'objet

Utilisation

```
switch (ace) {
case ROUGE: case ORANGE: case JAUNE :
System.out.println("couleur chaude"); break ;
case VERT : case BLEU : case INDIGO : case VIOLET :
System.out.println("couleur froide"); break;
default : System.out.println("???"); break;
```

On peut tester un objet de type énuméré comme un entier avec `switch`

Utilisation

```
for (ArcEnCiel a : ArcEnCiel.values()){System.out.println(a.name());}
```

On peut itérer sur toutes les valeurs du types énuméré par
`for (EnumClasse e : EnumClasse.values())`
la variable e prend successivement toutes les valeurs énumérées dans l'ordre d'énumération.

Ajout d'informations

Il est possible d'ajouter des informations propres à chaque valeur énumérée sous forme d'attributs ; on doit alors écrire constructeurs et méthodes correspondant à ces attributs.

Exemple

```
public enum ArcEnCiel2{
ROUGE(6.2E-7,"chaud"),
ORANGE(5.8E-7,"chaud"),
JAUNE(5.6E-7,"chaud"),
VERT(5.0E-7,"froid"),
BLEU(4.5E-7,"froid"),
INDIGO(4.0E-7,"froid"),
VIOLET(3.8E-7,"froid");
private final double l;
private final String chaleur;
private ArcEnCiel2(double d, String s){this.l=d;this.chaleur = s;}
public static String couleur(double d){
String s = "invisible";
if(d>ROUGE.l || d<VIOLET.l) return s;
for (ArcEnCiel2 a : ArcEnCiel2.values()){
if (d>a.l)
return( "couleur " + a.chaleur + " entre " + ArcEnCiel2.values()[a.ordinal()-1].name() +" et "+a
return s;                                }}
public class EssaiArcEnCiel2{
public static void main(String[] arg) {
double d= Double.parseDouble(arg[0]);
System.out.println(ArcEnCiel2.couleur(d));
}}
```

Chapitre IX – Interfaces – Classes abstraites

Chapitre IX – Interfaces – Classes abstraites

Chapitre IX – Interfaces – Classes abstraites

Chapitre IX – Interfaces – Classes abstraites

- I. Définitions des Interfaces
- II. L'interface : `Cloneable`
- III. L'interface : `Comparable`
- IV. Les classes abstraites

Définitions des Interfaces

Une interface est la description d'un ensemble de méthodes que les classes Java peuvent mettre oeuvre. Par nature les interfaces sont abstraites.

Une classe **implémente** une interface : chaque méthode de l'interface est implémentée dans la classe.

Cela peut être vu comme un contrat entre la classe et l'interface.

Une interface est définie au même niveau qu'une classe : elle contient

- des définitions de constantes
- des déclarations de méthodes (prototype uniquement).

Syntaxe :

interface MonInterface

class MaClasse **implements** MonInterface

Remarques :

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)

Définitions des Interfaces

Une interface est la description d'un ensemble de méthodes que les classes Java peuvent mettre oeuvre. Par nature les interfaces sont abstraites.

Une classe **implémente** une interface : chaque méthode de l'interface est implémentée dans la classe.

Cela peut être vu comme un contrat entre la classe et l'interface.

Une interface est définie au même niveau qu'une classe : elle contient

- des définitions de constantes
- des déclarations de méthodes (prototype uniquement).

Syntaxe :

```
interface MonInterface

class MaClasse implements MonInterface
```

Remarques :

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)

Définitions des Interfaces

Une interface est la description d'un ensemble de méthodes que les classes Java peuvent mettre oeuvre. Par nature les interfaces sont abstraites.

Une classe **implémente** une interface : chaque méthode de l'interface est implémentée dans la classe.

Cela peut être vu comme un contrat entre la classe et l'interface.

Une interface est définie au même niveau qu'une classe : elle contient

- des définitions de constantes
- des déclarations de méthodes (prototype uniquement).

Syntaxe :

interface MonInterface

class MaClasse **implements** MonInterface

Remarques :

- on peut placer les modificateurs **public** ou **abstract** avant le mot **interface**,
- une classe peut implémenter plusieurs interfaces différentes. Cela permet de contourner le problème de l'héritage multiple qui n'est pas permis en Java pour les classes, mais
- une interface peut étendre plusieurs autres interfaces (héritage multiple)

L'interface : `Cloneable`

La méthode d'instance `clone()` de la classe `Object` effectue une opération de clonage spécifique.
Si la classe de l'objet n'implémente pas l'interface Cloneable alors une `CloneNotSupportedException` est levée.
Sinon, une nouvelle instance de la classe `Object` est créée avec les attributs initialisés avec ceux de l'objet cloné : *la méthode clone de la classe `Object` duplique tous les attributs d'une classe et renvoie une instance `Object`*.
Par exemple, si une classe A contient :

- un attribut entier n
- un attribut t référençant un tableau,

la méthode `clone` de la classe `Object` appliquée à une instance a de A construit une nouvelle instance a2 de `Object` avec :

- l'entier n qui a, au moment de la construction de la copie, la même valeur que l'attribut n de l'instance a ; si on change la valeur de n dans la copie, on ne change pas la valeur de n dans l'original ;
- l'attribut t qui référence le même tableau l'attribut t de l'instance a

Exemple

```
class EssaiClone implements Cloneable{
int n;
int[] t = {1,2,3};

public Object clone() throws CloneNotSupportedException {
return super.clone();}

public static void main(String[] arg) throws CloneNotSupportedExcept
                EssaiClone o= new EssaiClone();
                o.n=0;
                EssaiClone oc= (EssaiClone)o.clone();
                System.out.println(oc.n+" "+ oc.t);
}}
```

Exécution : 0 [I@eb42cbf

Exemple

```
class EssaiClone implements Cloneable{
int n;
int[] t = {1,2,3};

public Object clone() throws CloneNotSupportedException {
return super.clone();}

public static void main(String[] arg) throws CloneNotSupportedExcept
                EssaiClone o= new EssaiClone();
                o.n=0;
                EssaiClone oc= (EssaiClone)o.clone();
                System.out.println(oc.n+" "+ oc.t);
}}
```

Exécution : 0 [I@eb42cbf

copie de surface/en profondeur

La méthode `clone()` réalise une copie de surface (shallow copy) : les références des attributs de type non primitifs sont copiés.
Pour réaliser une copie en profondeur (deep copy), on doit :

- récupérer l'objet à renvoyer en appelant la méthode super.clone() (copie de surface),
- cloner les attributs non immuables afin de passer d'une copie de surface à une copie en profondeur de l'objet.

Exemple

```
class EssaiCloneProfondeur implements Cloneable{
int n;
int[] t = {1,2,3};

public Object clone() throws CloneNotSupportedException {
EssaiCloneProfondeur o =(EssaiCloneProfondeur) super.clone();// cast
 o.t = new int[this.t.length];
for (int k=0;k<this.t.length;k++) o.t[k]=this.t[k];
return o;}

public static void main(String[] arg) throws CloneNotSupportedExcept
EssaiCloneProfondeur o= new EssaiCloneProfondeur();
o.n=0;
EssaiCloneProfondeur oc= (EssaiCloneProfondeur)o.clone();
oc.t[0]=12;
System.out.println("oc.n="+oc.n+"o.n="+o.n+"  oc[0]="+
        oc.t[0]+"   o[0] ="+ o.t[0]);
}}
```

Remarque : on peut changer le prototype de `clone()` :

public EssaiCloneProfondeur clone() ...

requis de la méthode clone()

- `x.clone() != x ;` doit renvoyer `true`
- `x.clone().getClass() == x.getClass() ;` doit renvoyer `true`
- `x.clone().equals(x) ;` doit renvoyer `true`

requis de la méthode clone()

- `x.clone() != x ;` doit renvoyer `true`
- `x.clone().getClass() == x.getClass() ;` doit renvoyer `true`
- `x.clone().equals(x) ;` doit renvoyer `true`

requis de la méthode clone()

- `x.clone() != x ;` doit renvoyer `true`
- `x.clone().getClass() == x.getClass() ;` doit renvoyer `true`
- `x.clone().equals(x) ;` doit renvoyer `true`

L'interface : `Comparable`

L'interface `Comparable` du le paquetage `java.lang` permet de définir une méthode de tri sur toute classe d'objets que l'on peut ordonner selon un ordre total (deux objets quelconques sont toujours comparables) et de façon transitive (si un objet a est avant un objet b lui-même avant un objet c alors l'objet a est avant l'objet c).
Cela permet d'utiliser les méthodes de tri standard en Java.
L'interface java.lang.Comparable est définie ainsi

```
public abstract interface Comparable {public int compareTo (Object obj);}
```

Cette méthode renvoie 0 en cas « d'égalité » -1 si l'objet considéré est avant le paramètre obj et $+1$ sinon.

Exemple

```
class EssaiCloneCompProfondeur implements Cloneable, Comparable{
int n;
int[] t = {1,2,3};
// voir EssaiCloneProfondeur
int compTableau(int[] tab) {
for (int k=0;k<this.t.length;k++) {if (tab[k]<this.t[k]) return -1;
else {if (tab[k]>this.t[k]) return 1; };}
return 0;
}
public int compareTo (Object obj){
if (((EssaiCloneCompProfondeur)obj).n< this.n) return 1;
else {if (((EssaiCloneCompProfondeur)obj).n> this.n) return -1;
else return this.compTableau(((EssaiCloneCompProfondeur) obj).t);}
}}
```

Application : pour trier un tableau de `EssaiCloneCompProfondeur` on utilise la méthode statique sort qui se trouve dans la classe java.util.Arrays de prototype **public static void** sort(Object[] tableau).

Les classes abstraites

Une classe est abstraite si

- elle est marquée par le modificateur `abstract`
- elle contient des (au moins 1) méthodes abstraites.

Une méthode abstraite

- est marquée par le modificateur `abstract`
- se déclare seulement par son prototype.

Elle n'est pas instanciable.
Une classe est abstraite peut être étendue par une classe qui devra alors définir toutes les méthodes abstraites héritées pour pouvoir, elle, être instanciée.
Cette notion est utile pour factoriser du code et laisser des méthodes abstraites qui peuvent être implémentées dans des sous-classes.

Exemple

On définit une classe abstraite `Quadrilatère` avec

- 4 attributs pour les longueurs des 4 côtés,
- une méthode d'instance `périmètre` renvoyant le périmètre d'un objet
- une méthode abstraite `surface` qui devra renvoyer la surface d'un objet.

.

Puis on définira des sous-classes Trapèze, Rectangle qui implémenteront la méthode `surface` différement mais qui pourront utiliser la méthode `périmètre` de leur super classe.

```
public abstract class Quadrilatere {
double a,b,c,d;
public abstract double surface();// prototype
public  double perimetre(){return (this.a+this.b+this.c+this.d)};}
public class Trapèze extends Quadrilatere {
double h;
public  double surface(){return (h*(b+d)/2);};
}
public class Rectangle extends Quadrilatere {
public  double surface(){return (a*b)};
}
```

Chapitre X – Les collections

Chapitre X – Les collections

Chapitre X – Les collections

Chapitre X – Les collections

Chapitre X – Les collections

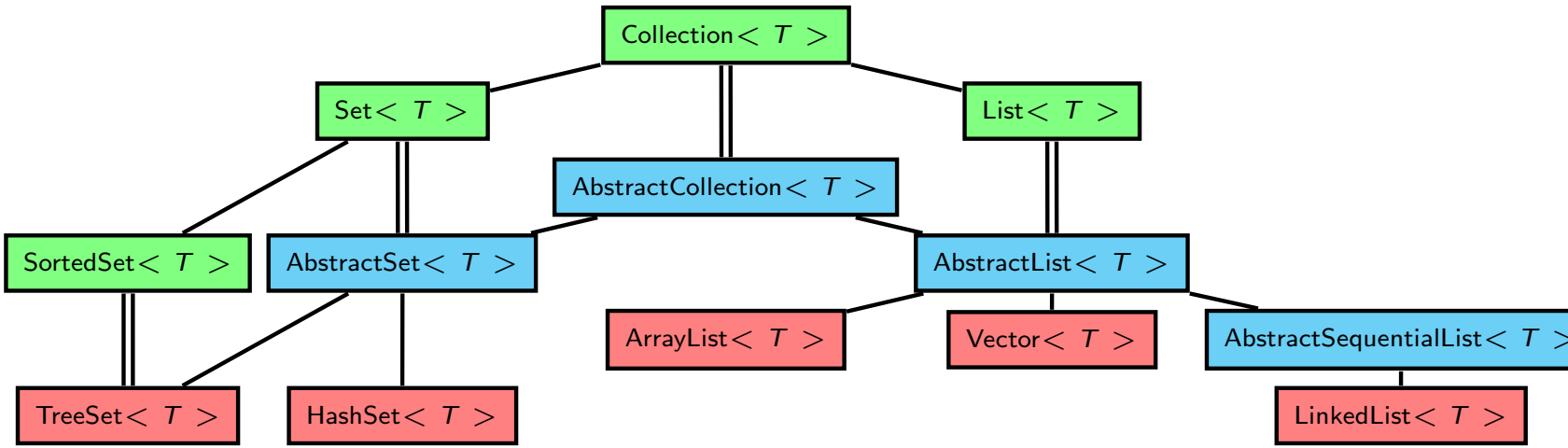
Java propose plusieurs moyens de manipuler des ensembles d'objets : on a vu les tableaux dont l'inconvénient est de ne pas être dynamique vis à vis de leur taille.
Java fournit des interfaces qui permettent de gérer des ensembles d'objets dans des structures qui peuvent être parcourues.
Ce chapitre donne un petit aperçu de ces collections.
Toutes les collections d'objets

- sont dans le paquetage *java.util*
- implémentent l'interface générique *Collection*

L'interface Set<> sert à implémenter les collections de type ensemble : les éléments n'y figurent qu'une fois et ne sont pas ordonnés.
L'interface List<> sert à implémenter les collections dont les éléments sont ordonnées et qui autorisent la répétition.

Interface Collection

les interfaces sont en vert, les classes abstraites en bleu et les classes en rouge, et T est le type paramètre des éléments des collections ; les lignes simples indiquent l'héritage et les lignes doubles l'implémentation.



Les méthodes

- `boolean add(Object)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(Object)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(Object)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(Object)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(Object)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(Object)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(Object)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(Object)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(Object)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(Object)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(Object)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(Object)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes

- `boolean add(Object)` ajoute l'élément fourni en paramètre à la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean addAll(Collection)` ajoute à la collection tous les éléments de la collection fournie en paramètre
- `void clear()` supprime tous les éléments de la collection
- `boolean contains(Object)` indique si la collection contient au moins un élément identique à celui fourni en paramètre
- `boolean containsAll(Collection)` indique si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
- `boolean isEmpty()` indique si la collection est vide
- `Iterator iterator()` renvoie un objet qui permet de parcourir l'ensemble des éléments de la collection

Les méthodes - suite

- `boolean remove(Object)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection

Les méthodes - suite

- `boolean remove(Object)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection

Les méthodes - suite

- `boolean remove(Object)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection

Les méthodes - suite

- `boolean remove(Object)` supprime l'élément fourni en paramètre de la collection. La valeur de retour indique si la collection a été mise à jour
- `boolean removeAll(Collection)` supprime tous les éléments de la collection qui sont contenus dans la collection fournie en paramètre
- `int size()` renvoie le nombre d'éléments contenu dans la collection
- `Object[] toArray()` renvoie d'un tableau d'objets qui contient tous les éléments de la collection

La classe Collections

La classe `java.util.Collections` (notez le pluriel) contient des méthodes *statiques* qui opérent sur des objets `List` ou autre (`Set`, `Map` ...) ou bien renvoie des objets.

- `void sort(List list)` trie le paramètre list
- `void sort(List list, reverseOrder() )` trie le paramètre list en ordre décroissant
- `Object max(Collection coll)` renvoie le plus grand objet
- `Object min(Collection coll)` renvoie le plus petit objet
- ...

On peut utiliser ces méthodes statiques sur des objets de toutes les classes du schéma précédent.

Itérateurs

Un iterateur est un objet utilisé avec une collection pour fournir un accès séquentiel aux éléments de cette collection.
L'interface `Iterator` permet de fixer le comportement d'un itérateur.

- `boolean hasNext()` indique s'il reste au moins un élément à parcourir dans la collection
- `Object next()` renvoie le prochain élément dans la collection
- `void remove()` supprime le dernier élément parcouru (celui renvoyé par le dernier appel à la méthode next())

La méthode `next()` lève une exception de type `NoSuchElementException` si elle est appelée alors que la fin du parcours des éléments est atteinte.
La méthode `remove()` lève une exception de type `IllegalStateException` si l'appel ne correspond à aucun appel à `next()`.
On ne peut pas faire deux appels consécutifs à `remove()`.

Interface ListIterator

L'interface `ListIterator<T>` étend l'interface `Iterator<T>` et permet de parcourir la collection dans les deux sens.

- `T previous()` renvoie l'élément précédent dans la collection
- `boolean hasPrevious()` teste l'existence d'un élément précédent
- `T next()` renvoie l'élément suivant de la liste
- `T previous()` renvoie l'élément précédent de la liste
- `int nextIndex()`
- `int previousIndex()`

Classe ArrayList

Un `ArrayList` est un tableau d'objets dont la taille est dynamique. La classe `ArrayList` implémente en particulier les interfaces `Iterator`, `ListIterator` et `List`.

Constructeurs

- `public ArrayList(int initialCapacite)` crée un `arrayList` vide avec la capacité `initialCapacite` (positif)
- `public ArrayList()` crée un `arrayList` vide avec la capacité 10
- `public ArrayList(Collection<? extends T> c)` crée un `arrayList` contenant tous les éléments de la collection c dans le même ordre avec une dimension correspondant à la taille réelle de c et non sa capacité ; le `arrayList` créé contient les références aux éléments de c (copie de surface).

Méthodes

- `add` et `addAll` ajoute à la fin du tableau
- `void add(int index, T element)` ajoute au tableau le paramètre element à l'indice index en décalant d'un rang vers la droite les éléments du tableau d'indice supérieur
- `void ensureCapacity(int k)` permet d'augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
- `T get(int index)` renvoie l'élément du tableau dont la position est précisée
- `T set(int index, T element)` renvoie l'élément à la position index et remplace sa valeur par celle du paramètre element

Méthodes

- `int indexOf(Object o)` renvoie la position de la première occurrence de l'élément fourni en paramètre
- `int lastIndexOf(Object o)` renvoie la position de la dernière occurrence de l'élément fourni en paramètre
- `T remove(int index)` renvoie l'élément du tableau à l'indice index et le supprime décalant d'un rang vers la gauche les éléments d'indice supérieur
- `void removeRange(int j,int k)` supprime tous les éléments du tableau de la position i incluse jusqu'à la position k exclue
- `void trimToSize()` ajuste la capacité du tableau sur sa taille actuelle

Exemple

On veut gérer un ensemble de personnes connaissant leur age, poids et taille par ordre de risque décroissant de problème cardiaque compte tenu de ces données.
On définit

- la classe `Personne` (nom, prenom)
- la classe `PersonneMedicalise` (étend `Personne` avec age, taille, poids, implémente l'interface `Comparable`)
- la classe `EnsPersonneMedicale` qui utilise `ArrayList`.

```
public class PersonneMedicalise extends Personne implements Comparable {
....
public int compareTo(Object p){
if (this.getAge()> (PersonneMedicalise)p.getAge()) return -1; else
if (this.getAge()< (PersonneMedicalise)p.getAge()) return 1; else
if (this.getPoids()> (PersonneMedicalise)p.getPoids()) return -1; else
if (this.getPoids()< (PersonneMedicalise)p.getPoids()) return 1; else
return 0;
}}

import java.util.*;
public class EnsPersonneMedicale {
ArrayList <PersonneMedicalise> e;
public EnsPersonneMedicale() {}
...
public  PersonneMedicalise quiEstEnDanger(){
Collections.sort(e);
return(e[0]);}
public int ageMoyen(){
Iterator <PersonneMedicalise> it = e.iterator();
int a=0;
while (it.hasNext()) a= a+ it.next().getAge();
if (e.size()>0) return (a/e.size());}}
```

Tout magasin de la marque Nisagam a un lieu de stockage dans laquelle 1000 objets maximum peuvent être entreposés. Chaque objet (item) est caractérisé par un code composé de 5 caractères numériques et par un prix de vente. Chaque objet peut être en plusieurs exemplaires dans le stock et la marque Nisagam propose un choix de 100 objets différents.

classe Item

Ecrire la classe `Item` avec

- ses attributs privés
- ses accesseurs en lecture et modification.

classe ItemEnStock

La classe `ItemEnStock` associe à chaque Item le nombre d'exemplaires présent en stock Ecrire la classe `ItemEnStock` avec

- ses attributs privés
- un constructeur prenant en paramètre un Item et un entier
- ses accesseurs en lecture et modification.

classe Stock

La classe `Stock` est caractérisée par

- le nom du magasin
- l'ensemble des Items en stock dans le magasin
- le nombre total d'Items en stock dans le magasin

Ecrire la classe `Stock` avec

- ses attributs privés
- ses accesseurs en lecture et modification.
- une méthode d'ajout d'une quantité donnée en paramètre d'un Item donné par son code
- une méthode de retrait d'une quantité donnée en paramètre d'un Item donné par son code
- une méthode qui renvoie la valeur totale du stock
- une méthode qui renvoie l'Item qui est le plus présent dans le stock

classe ChaineNisagam

On considère une chaîne de 10 magasins Nisagam. On s'intéresse donc à gérer globalement les stocks de ces 10 magasins.
Ecrire la classe `ChaineNisagam` avec

- son ou ses attributs privés
- ses accesseurs en lecture et modification.
- une méthode d'ajout Stocker d'un Item donné par son code dans le magasin dont le stock est le moins chargé et qui renvoie le nom de ce magasin
- une méthode qui renvoie le nom du magasin qui a le moins de valeur marchande en stock
- une méthode de classe qui renvoie la valeur marchande totale des 10 stocks

Dans la méthode `main`, vous devrez

- créer un Stock pour un magasin Nisagam$i$ pour chaque $i \in \{0..9\}$
- créer une chaîne de 10 magasins Nisagam1, ... Nisagam9
- ajouter 5 exemplaires d'un Item
- afficher le nom de magasin qui a le moins de valeur marchande en stock
- afficher la valeur marchande totale des 10 stocks

Rappels :

• la classe `java.util.Random` contient la méthode `public int nextInt(int k)` qui renvoie un entier dans $\{0, \ldots, k-1\}$.

• la méthode `String` concat(`String` s) ajoute la chaîne de caractères s derrière la chaîne objet.

• la méthode static String valueOf(int i) renvoie la chaîne de caractères représentant l'entier en argument