Programmation Orientée Objet
Cours 1

Mohamed Nabil Saidi

msaidi@insea.ac.ma

9 Février 2012

Plan

- Présentation de Java
- Le paradigme objet
  - Classes
  - Objets
- L'essentiel du langage de Java
  - Variables
  - Opérateurs
  - Commandes basiques

Présentation de Java

JAVA c'est quoi ?

- Une technologie développé par SUN Microsystem^{TM} lancé en 1995
- Un langage de programmation
- Une plateforme, environnement logiciel dans lequel les programmes java s'exécutent
- Présent dans de très nombreux domaines d'application : des serveurs d'applications aux téléphone portables et cartes à puces (JME)

Définition de Sun :

Java est un langage simple, orienté objet, distribué, robuste, sûr, indépendant des architectures matérielles, portable, de haute performance, multithread et dynamique

- Simple & Robuste
  - Abandonner les éléments mal compris ou mal exploites dans d'autres langages
    - Pas de pointeur
    - Pas d'héritage multiple
  - Typage des données très strict
- Sûr
  - Java n'est pas compilé à destination d'un processuer particulier mais en "byte code" qui pourra être ensuite interprété sur une machine virtuelle (JVM = Java Virtual Machine). Le "byte code" généré est vérifié par les interpréteurs java avant exécution.
  - Un débordement de tableau déclenchera automatiquement une exception.
  - L'absence d'arithmétique de pointeur évite les malversations.

Généralités

- Portable
  - Les types de données sont indépendants de la plate forme (par exemple les types numériques sont définis indépendamment du type de plate forme sur laquelle le byte code sera interprétée).
- Multi thread
  - Une applications peut être décomposée en unités d'exécution fonctionnant simultanément
- Dynamique
  - Les classes Java peuvent être modifiées sans avoir à modifier le programme qui les utilise
- Politique
  - Java est actuellement totalement controlé par SUN.

Plusieurs Java

- Java EE : “Enterprise Edition”. Rajoute certaines API et fonctionalités pour les entreprises.
- Java ME : “Micro Edition”. Édition qui sert à écrire des applications embarquées
  - Ex. : téléphone portable, carte à puce
- Java SE : “Standard Edition” :
  - JRE : “Java Runtime Environment”. Contient la plate-forme Java (JVM + API).
  - JDK : (“Java Development Kit”). Contient le langage de programmation et la plate-forme (compilateur + JVM + API).

Les différentes version de java

- Java 1.0
  - 8 packages
  - 212 Classes et Interfaces
  - 1545 Méthodes
- Java 1.1
  - 23 packages
  - 504 Classes et Interfaces
  - 3 851 Méthodes
- Java 1.2
  - 60 packages
  - 1 781 Classes et Interfaces
  - 15 060 Méthodes
- Et bien plus encore dans les versions suivantes

L'environnement de travail

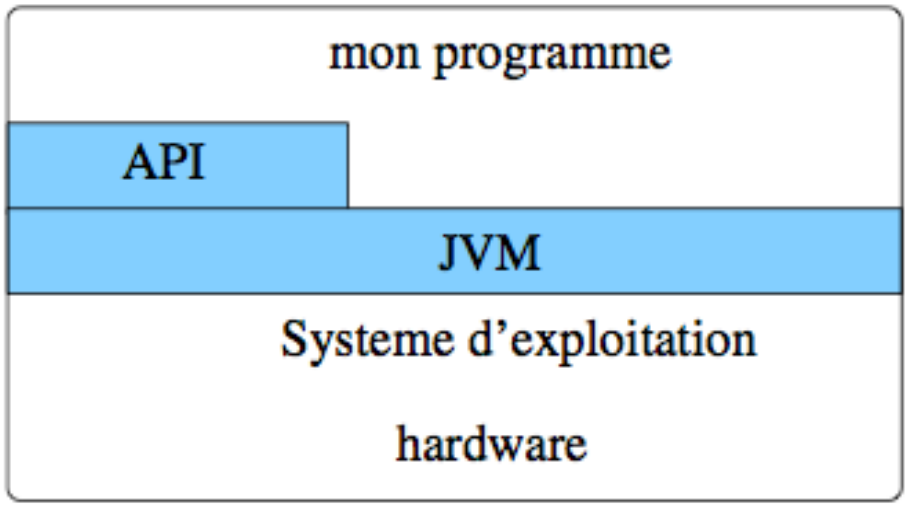
- Un éditeur de texte Ex. : Emacs, gedit, GVim
- Le compilateur (javac)
- La JVM (java)
- Le générateur automatique de documentation (javadoc)
- Le debogueur (jdb)
- La documentation du JDK 6, disponible à : http://java.sun.com/javase/6/docs/

Plate-forme Java

- La plate-forme est le matériel (“hardware”) et/ou l’environnement logiciel dans lequel un programme s’exécute.
- La plate-forme Java est un environnement logiciel, composée de deux parties :
  - API : (“Application Programming Interface “) grande collection de composants logiciels qui offrent de nombreuses fonctionnalités.
  - JVM : (“Java Virtual Machine”) le logiciel qui interprète le bytecode généré par le compilateur Java.

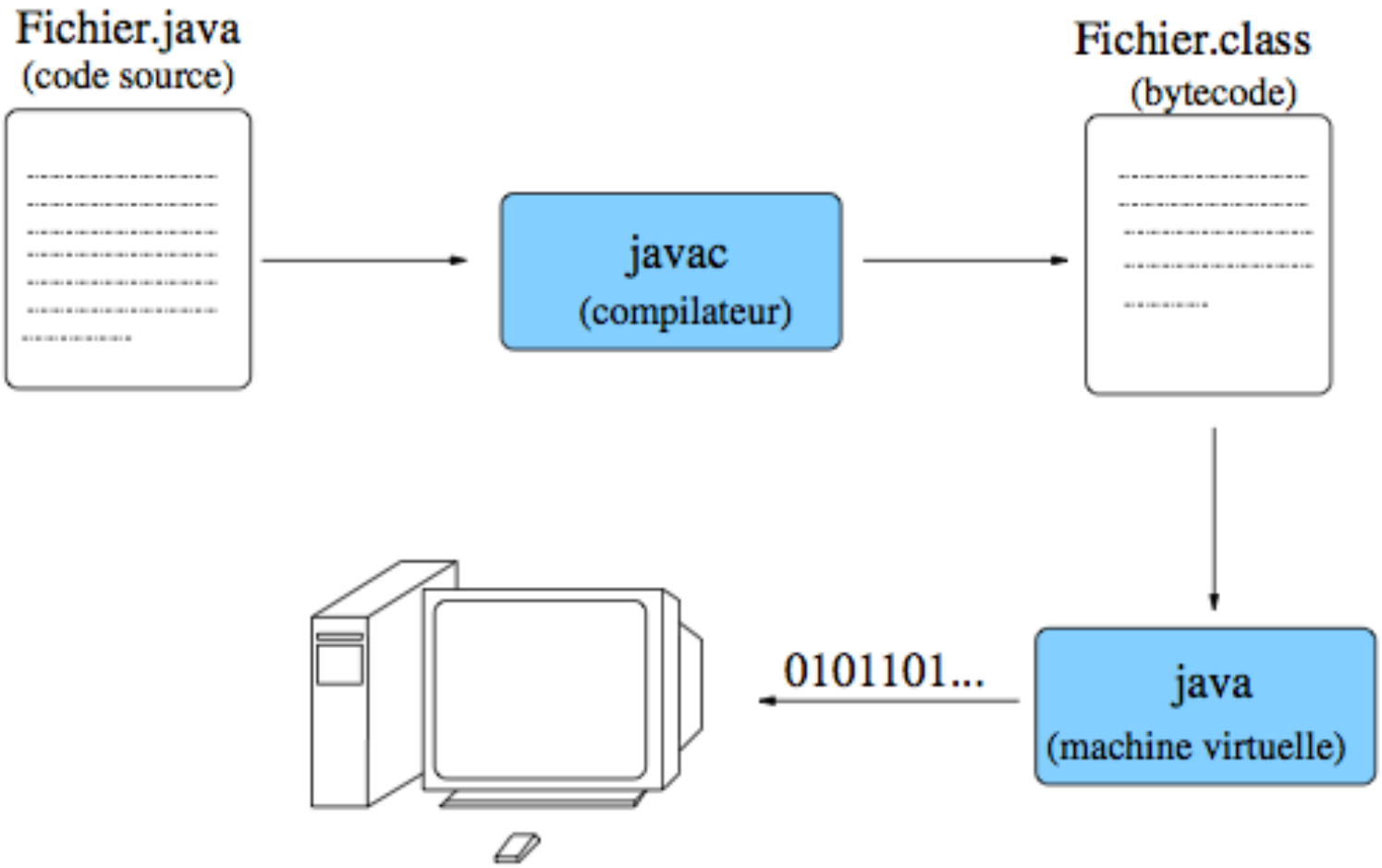
Plate-forme Java (2)

- Un programme Java est exécuté par la JVM, qui s'utilise de l'API.



Plate–forme Java

Compilation et exécution



La machine virtuelle

- Il y a des JVM pour la plupart des systèmes
  - Ex. : Windows, Linux, Mac OS, Solaris
- Si un système possède une JVM, il peut exécuter le bytecode généré sur n'importe quel autre système.
- Avantages de cet approche :
  - Portabilité : le bytecode peut être chargé depuis une machine distante sur Internet.
  - Sécurité : la JVM effectue des nombreuses vérifications sur le bytecode pour éviter des actions "dangereuses".

La machine virtuelle (2)

- Désavantage de cet approche : lenteur.
- Mais, des nouvelles techniques essayent de minimiser ce problème :
  - Ex., la traduction en code binaire des parties du bytecode qui sont utilisés très fréquemment.

Utilisation de JAVA

- Le Langage java peut générer
  - des applications
  - des applets
  - des servelets
  - etc.

Génération du code exécutable

- Le code est généré par un compilateur en plusieurs étapges :
  - Vérification syntaxique.
  - Vérification sémantique (typage).
  - Production de code dans un langage plus proche de la machine

Production de code

- Avantages/inconvinients du code natif
  - Rapidité d'exécution
  - Nécessité de recompiler lors du portage d'un logiciel sur une autre architecture/système d'exploitation
  - Choix de la distribution du logiciel: source ou binaire ?
  - Java, production de code intermédiaire: le bytecode

Variables d'environnement

- Les variables d'environnement doivent être correctement initialisées:
  - `CLASSPATH` répértoire contenant les classes.
  - `JAVA_HOME` répértoire de base du JDK
  - `PATH` répértoire contenant le compilateur et l'interpréteur

Avantages/Inconvinients du bytecode

- Code portable au niveau binaire
- Moins efficace que du code natif

Un programme en Java

Code source (dans un fichier texte) :

HelloWorld.java

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

Le main()

- Le point d'entrée pour l'exécution d'une application Java est la méthode statique **main** de la classe spécifiée à la machine virtuelle
- Profil de cette méthode
  - **public static void main(String [] args)**
- **String args** ???
  - args : tableau d'objets String (chaînes de caractères) contenant les arguments de la ligne de commande

Le paradigme objet

Paradigmes de programmation

- Un paradigme de programmation correspond à une manière de modéliser le monde.
- Il existent plusieurs paradigmes :
  - programmation **impérative** (ex. : Pascal, C, Fortran) ;
  - programmation **fonctionnelle** (ex. : Scheme, Lisp) ;
  - programmation **logique** (ex. : Prolog) ;
  - programmation **orientée objet** (ex. : C++, Java).
- Dans le paradigme objet :
  - Le monde est modélisé comme un ensemble d'objets.
  - Les objets ont un état interne et un comportement.
  - Ils collaborent en s'échangeant des messages.

Qu'est-ce qu'un objet ?

- **Toute entité identifiable, concrète ou abstraite.**
  - Ex. : stylo, table, ordinateur, vélo, logiciel.
- Deux caractéristiques importantes :
  - État
  - Comportement
- L'objet vélo :
  - États : vitesse, couleur, direction, etc.
  - Comportements : accélérer, s'arrêter, tourner à droite, etc.

Concepts des langages objet

- Concept de base de la programmation orientée objet : la classe
- Une classe modélise la structure statique (données membres) et le comportement dynamique (méthodes) des objets associés à cette classe.
- Un objet d'une classe est appelé une instance.
- Une classe est la description d'un objet. Chaque objet est créé à partir d'une classe (avec l'opérateur **new**).

Notion d'objet en programmation

Un objet a :

- une **identité** : adresse en mémoire
- un **état** : la valeur de ses attributs
- un **comportement** : ses méthodes

Classes d'objets

- Une définition abstraite selon laquelle les objets sont crées (un type d'objet)
  - Ex. : la classe des vélos, la classe des stylos
- **Exemple** : Définir une classe Velo :
  - Attributs : vitesse
  - Mthodes : Accélérer, Freiner et ImprimeEtat

Définition d'une classe

Velo.java

```
class Velo {
int vitesse = 0;
void accelerer(int increment) {
vitesse = vitesse + increment;
}
void freiner(int decrement) {
vitesse = vitesse - decrement;
}
void imprimeEtat() {
System.out.println("vitesse: " + vitesse);
}
}
```

Création des objets et messages

DemoVelo.java

```
class DemoVelo {
public static void main(String[] args) {
// Genere deux objets differents du type Velo
Velo velo1 = new Velo();
Velo velo2 = new Velo();
// Invoque les methodes
velo1.accelerer(10);
velo1.imprimeEtat();
velo2.accelerer(20);
velo2.imprimeEtat();
}
}
```

Un programme en Java (2)

Compilation (dans la console) :

$ javac HelloWorld.java

Le compilateur recoit un nom d'un fichier ayant pour suffixe .java. Ensuite, il génère le bytecode dans un fichier ayant pour suffixe .class. Exécution (dans la console) :

$ java HelloWorld
Hello World!

Attention : La JVM recoit un nom d'une classe (donc, pas de suffixe .class). Le fichier contenant le bytecode de la classe doit être présent dans le même dossier.

Compilation et exécution

- Compilation :

  $ javac Velo.java DemoVelo.java

- Exécution :

  $ java DemoVelo
  vitesse: 10
  vitesse: 20

- Un programme source Java correspond à plusieurs fichiers .java.
- Chaque fichier .java peut contenir une ou plusieurs définitions de classes.

Héritage

- Il existent plusieurs types de vélos.
  - Ex. : vtt, vélo route, vélo ville
- Ils ont plusieurs caractéristiques communes.
- Mais, certaines caractéristiques sont propres à un type spécifique.
  - Ex. : un vélo ville a un garde bout
- Il est possible de créer une **sous-classe** qui hérite l'état et le comportement d'une **superclasse**.
- Cela permet de réutiliser le code de la superclasse.

Hiérarchie de classes

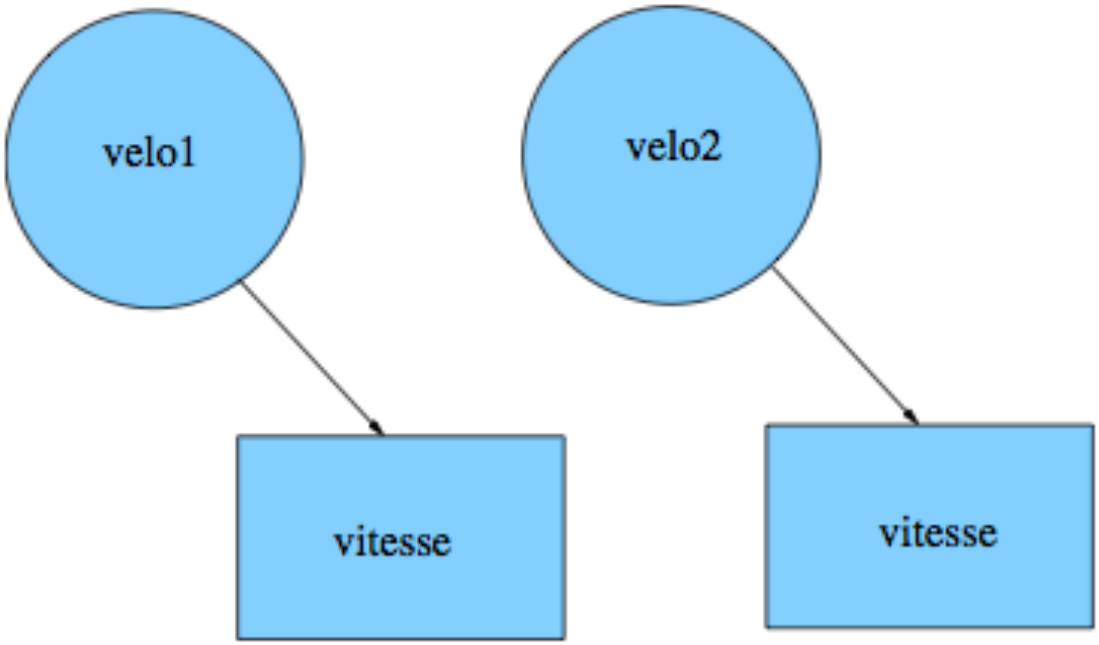
```
class VeloRoute extends Velo
{
// Nouvelles variables et méthodes ...
}
```

Objets

- Un objet est une **instance** d'une (seule classe) :
  - il se conforme à la description que celle-ci fournit,
  - il admet une valeur (**qui lui est propre**) pour chaque attribut déclaré dans la classe,
  - ces valeurs caractérisent **l'état** de l'objet
  - il est possible de lui appliquer toute opération (**méthode**) définit dans la classe
- Tout objet admet une identité qui le distingue pleinement des autres objets:
  - il peut être nommé et être **référencé** par un nom

Chaque objet vélo instance de la classe Vélo possédera sa propre vitesse.

Représentation mémoire



Références

- Pour désigner des objets dans une classe (attributs ou variables dans le corps d'une méthode) on utilise des variables d'un type particulier: **les références**
- Une référence contient l'adresse d'un objet
  - pointeur vers la structure de données correspondant aux attributs (variables d'instance) propres à l'objet.
- Une référence peut posséder la valeur null
  - aucun objet n'est accessible par cette référence
- Déclarer une référence ne crée pas d'objet
  - une référence n'est pas un objet, c'est un nom pour accéder à un objet

Références

- **Comme un pointeur** une référence contient l'adresse d'une structure
- Mais **à la différence des pointeurs** la seule opération autorisée sur les références est **l'affectation** d'une référence de même type

```
Velo v1;
...
Velo v2;
v2 = v1;
v1++; Erreur
...
v2 += *v1 + 3;
```

Créations d'Objets

- **new constructeur (liste de paramètres)**
- les constructeurs ont le même nom que la classe
- il existe un constructeur par défaut

```
Velo v1;
v1 = new Velo();
Velo v2 = new Velo();
Velo v3 = v2;
```

Constructeur de classe

- Un **constructeur** est une méthode automatiquement appelée au moment de la création de l'objet.
- Un constructeur est utile pour procéder à toutes les initialisations nécessaires lors de la création de la classe.
- Le constructeur porte le même nom que le nom de la classe et n'a pas de valeur de retour.

Exemple de constructeur

```
class Compte
{int numero;
float solde;
public Compte (int num,float s)
{
numero = num ;
solde = s ;
}
}
Compte co1 = new Compte (1234, 1000.00f) ;
```

Représentation mémoire

- vélo1 et vélo2 contiennent l'adresse des zones mémoires allouées par l'opérateur new pour stocker les informations relatives à ces objets.
- vélo1 et vélo2 sont des références.
- La référence d'un objet est utilisée pour accéder aux données et fonctions membres de l'objet.
- Un objet peut accéder à sa propre référence grâce à la valeur **this** (variable en lecture seule).
- Une référence contenant la valeur null ne désigne aucun objet.
- Quand un objet n'est plus utilisé (aucune variable du programme ne contient une référence sur cet objet), il est automatiquement détruit et la mémoire est récupérée (garbage collector).

Destructeur de classe

- Un destructeur peut être utilisé pour libérer les ressources spécifiques (déconnexion d'une base de données, fermeture d'un fichier, ...).
- Il est appelé lorsque le garbage collector récupérera la mémoire.
- Un destructeur est une méthode:

void finalize ()

Surcharge de méthodes

- Une méthode (y compris le constructeur) peut être définie plusieurs fois avec le même nom à condition de se différencier par le nombre et/ou le type des paramètres transmis (polymorphisme).
- Le compilateur décidera de la bonne méthode à utiliser en fonction des paramètres d'appel.

Exemples de surcharge de méthodes

Exemple:

```
class BarreDeProgression
{
float pourcent ;

public void setPourcent (float valeur) { pourcent = valeur ;}
public void setPourcent (int effectue, int total)
{
pourcent = total/effectue ;
}

}
```

Types de variable

- Attributs : (ou **variables membres de classe**) variables définies dans une classe (à l'extérieur de ses méthodes).
  - Attribut d'instance : sa valeur est différente pour chaque instance de la classe (défaut).
  - Attribut de classe : sa valeur est la même pour toute instance de la classe.
- Variables locales : variables définies dans le corps de déclaration d'une méthode. (Visible seulement à l'intérieur de la méthode correspondante.)
- Paramètres : variables définies dans la liste des paramètres d'une méthode. (Visibles seulement à l'intérieur de la méthode correspondante.)

Accès aux attributs d'un objet

- pour accéder aux attributs d'un objet on utilise une notation pointée

  nomDeObjet.nomDeVariableDinstance

- similaire à celle utilisée en C pour l'accés aux champs d'une structure (Struct)

```
Velo v1;
v1 = new Velo();
Velo v2 = new Velo();
Velo v3 = v2;
v1.vitesse = 10;
v2.vitesse = 10;
v3.vitesse = v1.vitesse + v2.vitesse;
```

Envoi de messages (exemple)

- syntaxe :
  - **nomDeObjet.nomDeMethode(< paramètre effectifs >)**

```
class Velo {
double vitesse = 110.0;
void accelerer(double dx)
{
vitesse += dx;
}
void freiner(double dx){
vitesse -= dx;
}
} //Velo
```

Velo v1 = new Velo();
Velo v2 = new Velo();

v1.accelerer(10.0);
v2.freiner(20.0);

System.out.println("vitesse de v1 est de "+v1.**vitesse()**);

Exercice

- Ecrire une classe Compte contenant :
  - Attributs : Numéro du compte et solde
  - Méthodes : initialiser, deposer, retirer, consulterSolde et Afficher.
- Ecrire une classe Banque qui gére plusieurs objets de type Compte

Ecriture de la classe Compte

```
class Compte {
int numero ;
float solde ;
void initialiser (int n, float s) { numero = n ; solde = s ; }
void deposer (float montant) { solde = solde + montant ; }
void retirer (float montant) { solde = solde - montant ; }
float consulterSolde ( ) { return solde ; }
void afficher()
{ System.out.println ("Compte : " + numero + " solde: " +
solde) ;
}
}
```

Utilisation de la classe Compte

```
public class Banque
{ static public void main (String args [])
{
Compte co1 = new Compte () ;
Compte co2 = new Compte () ;
co1.initialise (1234,1000f) ; co2.initialise (5678,500f) ;
co1.deposer (2100.95f) ; co1.afficher () ;
co2.retirer (1000.0f) ; co2.afficher () ;
}
}
```

Syntaxe de base

Syntaxe de base du langage Java

- Les commentaires existent sous plusieurs formes:
  - Commentaires multi lignes
    - /*
    - */
  - Commentaires sur une seule ou fraction de ligne
    - //
  - Commentaires destinés au générateurs de documentation javadoc
    - /**
    - *
    - *
    - */

Type de données prédéfinis

- **byte** -2^7, (2^7)-1 -128, 127
- **short** -2^{15}, (2^{15})-1 -32768, 32767
- **int** -2^{31}, (2^{31})-1 -2147483648, 2147483647
- **long** -2^{63}, (2^{63})-1 -9223372036854775808, 9223372036854775807
- Les entiers peuvent être exprimés en octal (0323), en décimale (311) ou en hexadécimal (0x137).

Type de données prédéfinis

- Nombres réels
  - float simple précision sur 32 bits 1.4032984e-45 3.40282347 e38
  - double précision sur 64 bits 4.9406564584124354 4 e-324 1.7976931348623157 0 e308
  - Repésentation des réels dans le standard IEEE 754. Un suffixe " **f** " ou " **d** " après une valeur numérique permet de spécifier le type.
    - Exemple

```
double x = 154.56d;
float y = 23.4f;
float f = 23.65; //Erreur
```

Type de données prédéfinis

- boolean
  - Valeurs true ou false
  - Un entier non nul est également assimilé à true
  - Un entier nul est assimilé à false
- char
  - Une variable de type char peut contenir un seul caractère codé sur 16 bits (jeu de caractères 16 bits Unicode contenant 34168 caractères)
  - Des caractères d'échappement existent
    - \b Backspace \t Tabulation
    - \n Line Feed \f Form Feed
    - \r Carriage Return \" Guillemet
    - \' Apostrophe \\ BackSlash
    - \xdd Valeur hexadécimale \ddd Valeur octale
    - \u00xx Caractère Unicode (xx est compris entre 00 et FF)

Types primitifs

En Java, les types sont statiques (statically-typed language) et toute variable doit être déclarée avant son utilisation.

Type	valeurs possibles	valeur par défaut
byte entiers	8-bit	0
short entiers	16-bit	0
int entiers	32-bit	0
long entiers	64-bit	0L
float	virgule flottante 32-bit	0.0f
double	virgule flottante 64-bit	0.0d
boolean	true, false	false
char 16-bit	(caractère unicode)	'\ u0000'

Attention : les valeurs par défaut ne sont pas affectés aux variables locales !!

Déclaration et initialisation des variables

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
int decVal = 26; // Le numero 26, en decimal
int octVal = 032; // Le numero 26, en octal
int hexVal = 0x1a; // Le numero 26, en hexadecimal
double d1 = 123.4;
double d2 = 1.234e2; // notation scientifique
float f1 = 123.4f;
```

Types de données prédéfinis

- Chaînes de caractères
  - Les chaînes de caractères sont manipulées par la classe String (ce n'est donc pas un type de données).
- Exemples :

```
String str = "exemple de chaîne de caractères" ;
String chaine = "Le soleil " + "brille" ; //
Opérateur de concaténation
```

Chaînes de caractères

- Java fournit un support spécial aux chaînes des caractères.
- Exemple de déclaration :
  - **String s = "une chaîne de caractères";**
- La valeur initiale d'une variable du type String est null (ainsi que pour toutes les variables dont leur type est une classe d'objets).
- Mais attention!! Techniquement, String n'est pas un type primitif. Il s'agit d'une classe du paquetage java.lang.
- Strings sont des objets immuables ("immutable objects") : leur valeurs ne peuvent pas être modifiés.

Comparaison d'objets

- On ne peut pas comparer 2 objets en comparant les variables d'instance.
  - Exemple 1 :
    - r1 = new Rectangle (10,20) ;
    - r2 = new Rectangle (30,40) ;
    - r3 = new Rectangle (10,20) ;
    - Comparaison des variables d'instance:
    - r1 == r2 $\rightarrow$ false
    - r1 == r3 $\rightarrow$ false
    - Comparaison avec une méthode equals incluse dans la classe Rectangle
    - r1.equals (r2) $\rightarrow$ false
    - r1.equals (r3) $\rightarrow$ true

Exemple 2:

Comparaison de chaînes de caractères:
String s1 = "Bonjour" ;
String s2 = "Bonjour" ;

if (s1.equals (s2)) // *Compare le contenu de s1 et s2.*

if (s1.equalsIgnoreCase (s2)) // *Compare le contenu de s1 et s2*
// sans tenir compte des majuscules
// et minuscules.

Les tableaux

- Les tableaux peuvent être déclarés suivant les syntaxes suivantes :
  - type [] nom ;
- Exemples :
  - int[] table;
  - double [] d1,d2 ;
- Pas de tableau statique.
- La taille d'un tableau est allouée dynamiquement par l'opérateur new

$$\text{table} = \text{new int } [10] ;$$
$$\text{int[] table2} = \text{new int } [20] ;$$
$$\text{int[] table3} = \{1,2,3,4,5\} ;$$

Les tableaux

- La taille n'est pas modifiable et peut être consultée par la propriété length

```
System.out.println (table3.length) ;
int [ ] [ ] Matrice = new int [10][20] ;
System.out.println (Matrice.length) ; // 1ère dimension
System.out.println (Matrice[0].length) ; // 2ème dimension
```

Tableaux (arrays)

DemoTableau.java

```
class DemoTableau {
public static void main(String[ ] args) {
int[ ] unTableau; // declaration
unTableau = new int[3]; // allocation de memoire
unTableau[0] = 100; // initialisation
unTableau[1] = 200;
unTableau[2] = 300;
System.out.println(" Element 0: " + unTableau[0]);
System.out.println(" Element 1: " + unTableau[1]);
System.out.println(" Element 2: " + unTableau[2]);
}
}
```

Tableaux multidimensionnels

DemoMultiTableau.java

```
class DemoMultiTableau {
public static void main(String[] args) {
String[][] noms = {{"Mr. ", "Mrs. ", "Ms. "},
{"Smith", "Jones"}};
System.out.println(noms[0][0] + noms[1][0]);
System.out.println(noms[0][2] + noms[1][1]);
}
}
```

```
$ javac DemoMultiTableau.java
$ java DemoMultiTableau
Mr. Smith
Ms. Jones
```

Pour copier un tableau

Définition de la méthode (dans la classe System) :

public static void arraycopy (Object src, int posSrc, Object dest, int posDest int longueur)

Exemple d'utilisation :

DemoArrayCopy.java

```
class DemoArrayCopy {
public static void main(String[] args) {
char[] source = { 'd', 'e', 'c', 'a', 'f', 'e',
'i', 'n', 'e'};
char[] destin = new char[4];
System.arraycopy(source, 2, destin, 0, 4);
System.out.println(new String(destin));
}
}
```

Opérateurs

Par ordre de prioritée :

postifx	++ −−
unaires (prefixes)	++ −− + - ∼ !
multiplicatifs	* / %
additifs	+ -
décalage	>> << >>>
relationnels	< > <= >= instanceof
égalité	== !=
bitwise AND	&
bitwise excl. OR	^
bitwise incl. OR	|
conjonction	&&
disjonction	||
conditionnel	? :
affectation	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Opérateurs (2)

Quelques remarques :

- Opérateur $++$ (resp. $--$) préfixé évalue l'expression avant l'incrementation (resp. décrementation)
- Opérateurs && et || présentent le "short circuit behaviour" : le deuxième opérande est évalué seulement si nécessaire.
- Opérateur $+$ est utilisé aussi pour la concatenation des Strings.

L'opérateur instanceof

DemoInstanceof1.java

```
class DemoInstanceof1 {
public static void main(String[] args) {
Pere obj = new Pere();
System.out.println("obj instanceof Pere: " +
(obj instanceof Pere));
System.out.println("obj instanceof Fils : " +
(obj instanceof Fils));
}
}
class Pere {}
class Fils extends Pere {}
```

L'opérateur instanceof (2)

```
$ javac DemoInstanceof1.java
$ java DemoInstanceof1
obj instanceof Pere: true
obj instanceof Fils: false
```

L'opérateur instanceof (3)

DemoInstanceof2.java

```
class DemoInstanceof2 {
public static void main(String[] args) {
Fils obj = new Fils();
System.out.println("obj instanceof Pere: " +
(obj instanceof Pere));
System.out.println("obj instanceof Fils: " +
(obj instanceof Fils));
}
}
class Pere {}
class Fils extends Pere {}
```

L'opérateur instanceof (4)

```
$ javac DemoInstanceof2.java
$ java DemoInstanceof2
obj instanceof Pere: true
obj instanceof Fils: true
```

Commandes basiques

- Affectation :

  Variable OpAffectation Expression ;

- Bloc de commande :

  { Commande ; Commande ; ... ; }

- Contrôle de flux :

  if (Expression) Commande
  if (Expression) Commande else Commande
  switch (Expression) case ExpConstante :
  Commande ... [default : Commande] }
  while (Expression) Commande
  do Commande while (Expression) ;
  for (Commande ; [Expression] ; [Commande])
  Commande

Commandes basiques (2)

- Commandes de ramification :

  break [Identificateur] ;
  continue [Identificateur] ;
  return [Expression] ;

Exercices

- Ecrire un programme java demande deux valeurs numériques et qui affiche celui le plus grand
- Ecrire un programme java qui calcule le factoriel d'un nombre fournit en paramtre d'entrée

Programmation Orientée Objet
Cours2

Mohamed Nabil Saidi

msaidi@insea.ac.ma

16 Février 2012

Classes

- Composants de la déclaration d'une classe (dans l'ordre) :
  1. Modificateurs (optionnels)
  2. Mot-clé **class** suivie du nom de la classe (obligatoire)
  3. Mot-clé extends suivie du nom de la superclasse (optionnel)
  4. Mot-clé implements suivie d'une liste de noms d'interfaces (optionnel, expliqué plus tard)
  5. Corps de déclaration entouré par { et } public class Vtt extends Velo implements InterfaceVelo

```
// Declarations des attributs et methodes
```

Attributs

- Composants de la déclaration d'un attribut (dans l'ordre) :
  1. Modificateurs (optionnels)
  2. Type
  3. Nom

  - private int vitesse;

Les modificateurs d'accessibilité

A l'intérieur d'une classe, une variable ou une méthode peut être définie avec un modificateur d'accès. Les différents modificateurs d'accessibilité sont :

private l'élément (variable ou méthode, d'instance ou de classe) est privé, il n'est accessible que depuis la classe elle-même (le code de la classe dans laquelle il est défini);

pas de modificateur d'accès l'accès est dit *package*.

protected l'accès est étendu (par rapport à `private` au code des classes du même package **et** aux sous-classes de la classe. *Nous reviendrons plus tard sur cette notion, liée à l'héritage;*

public accessible à partir de tout code qui a accès à la classe où l'élément est défini.

Contrôle d'accès

- modificateur **public** : la classe est visible à toute autre classe ;
- pas de modificateur : la classe est visible seulement dans son paquetage.

Modificateurs d'accès aux attributs

- Le premier modificateur (plus à gauche) permet de contrôler l'accès à l'attribut :

modificateur	**classe**	**paquetage**	**sous-classe**	**autre**
public	oui	oui	oui	oui
protected	oui	oui	oui	non
aucun	oui	oui	non	non
private	oui	non	non	non

Choix du niveau d'accès

- Pour respecter le principe **d'encapsulation**, il est préférable d'utiliser private.
- Si nécessaire, l'accès à l'attribut par une autre classe sera fait indirectement, par le biais des méthodes (comme ex. les méthodes accelerer et freiner de la classe Velo).
- Les membres déclarés comme public ont tendance à relier le programme à une implémentation particuliére.

Attributs de classe

- Le modificateur static permet la création d'attributs de classe (aussi appelés attributs **statiques** et **variables de classe**).
- La valeur d'un attribut de classe est partagé par touts les objets de la classe.
- Tout objet de la classe peut changer sa valeur :

```
public class Velo {
...
private static int numVelos = 0;
...
public Velo(int vitesse) {
vitesse = 0;
++numVelos;
}
}
```

Attributs de classe

- Les attributs de classe peuvent être manipulés sans la création d'un objet !
  - ++Velo.numVelo;
- Il est possible de se référer à un attribut de classe en utilisant le nom de l'objet. Mais cela n'est pas conseillé, car il n'est pas claire qu'il s'agit d'un tel type d'attribut.

Membres statiques

- Déclaration est précédée du modifieur static
- variables de classe : définies et existent indépendamment des instances
- méthodes de classe : dont l'invocation peut être effectuée sans passer par l'envoi d'un message à l'une des instances de la classe.

accès aux membres statiques

- n'est pas conditionné par l'existence d'instances de la classe,
- directement par leur nom dans le code de la classe où ils sont définis,
  - en les préfixant du nom de la classe en dehors du code de la classe
  - NomDeLaClasse.nomDeLaVariable
  - NomDeLaClasse.nomDeLaMéthode(liste de paramètres)
  - Exemple : **Math.PI Math.cos(x) Math.toRadians(90)** ...

Les méthodes

On appelle signature dune méthode

- son nom,
- le type de ce quelle retourne,
- et la nature de ses arguments

Cest ce quon appelle parfois aussi son **entête**.

Les méthodes

- Déclaration d'une méthode (dans l'ordre) :
  - Modificateurs (optionnels)
  - Type de retour
  - Nom
  - Liste de paramètres typés entre parenthèses
  - Liste d'exceptions (optionnel, expliqué plus tard)
  - Corps de déclaration entouré par { et }

Surcharge (overloading)

- Les méthodes dans une classe peuvent avoir le même nom, si leurs signature est différente.
- La signature d'une méthode correspond à son nom et les types de ces paramètres.

Attention : la valeur de retour d'une méthode n'est fait pas partie de sa signature.

Constructeurs d'une classe :

- méthodes particulières pour la création d'objets de cette classe
- méthodes dont le nom est identique au nom de la classe
- rôle d'un constructeur
  - effectuer certaines initialisations nécessaires pour le nouvel objet créé
- toute classe JAVA possède au moins un constructeur
- si une classe ne définit pas explicitement de constructeur, un constructeur par défaut sans arguments et qui n'effectue aucune initialisation particulière est invoqué

Constructeurs

- Possibilité de définir plusieurs constructeurs dans une même classe
- possibilité d'initialiser un objet de plusieurs maniéres différentes : constructeurs multiples
- le compilateur distingue les constructeurs en fonction :
  - du nombre
  - du type
  - de la position des arguments

Constructeurs

```
public class Point {
private double x;private double y;
public Point(double x, double y) {
this.x = x; this.y = y;
} public Point() {
this.x = this.y = 0;
}
public Point(Point p) {
this.x = p.x;
this.y = p.y;
}
Point p1 = Point(10,10);
Point p2 =new Point();
Point p3 =new Point(p1);
```

Appel d'un constructeur par un autre constructeur

```
public class Point {
private double x; private double y;
// constructeurs
public Point(double x, double y)
{
this.x = x; this.y = y;
}
public Point(Point p) {
x = p.x; y = p.y;
}
public Point()
{
x = 0.0; y= 0.0;
}
...
}
```

Initialisation des variables la déclaration

- les variables d'instance et de classe peuvent avoir des "initialiseurs" associés à leur déclaration :
  modifieurs type nomDeVariable = expression;

```
private double x = 10;
private double y = x + 2;
private double z = Math.cos(Math.PI / 2);
private static int nbPoints = 0;
```

- variables de classe initialisées la premiére fois que la classe est chargée.
- variables d'instance initialisées lorsqu'un objet est créé.
- les initialisations ont lieu dans l'ordre des déclarations.

Destruction des objets

- libération de la mémoire alloué aux objets est automatique
- lorsqu'un objet n'est plus référencé le "ramasse miettes" ("**garbage collector**") récupère l'espace mémoire qui lui était réservé.
- s'exécute :
  - lorsqu'il n'y a pas d'autre activité (attente d'une entrée clavier ou d'un événement souris)
  - lorsque l'interpréteur JAVA n'a plus de mémoire disponible
- peut être moins efficace que la gestion explicite de la mémoire, mais programmation beaucoup plus simple et sûre.

Destruction des objets

```
class Point {
private double x;
private double y;
public Point(double x, double y)
{
this.x = x;
this.y = y;
}
public void translater(double dx, double dy)
{
x += dx;
y += dy;
} public String toString()
{ return "Point[x:" + x + ", y:" + y + "]";
}
public void finalize()
{
System.out.println("finalisation de " + this);
}
}
```

à refaire

Destruction des objets

```
Point p1 = new Point(14,14);
Point p2 = new Point(10,10);
System.out.println(p1);
System.out.println(p2);
p1.translater(10,10);
p1 = null;
System.gc(); //Appel explicite au garbage collector
System.out.println(p1);
System.out.println(p2);
```

```
Point[x:14.0, y:14.0]
Point[x:10.0, y:10.0]
finalisation de Point[x:24.0, y:24.0]
null
Point[x:10.0, y:10.0]
```

Types de données en Java

- 2 grands groupes de types de données :
  - **types primitifs**
  - **objets (instances de classe)**

- Java manipule différemment les valeurs des types primitifs et les objets : les variables contiennent
- des valeurs de types primitifs
- ou des références aux objets

Les variables de types primitifs

- **Caractérisation :** La déclaration d'une telle variable entraîne implicitement l'allocation de l'espace mémoire nécessaire à mémoriser une grandeur du type associé.
- Exemple : La déclaration d'une variable entière int n ; alloue les 32 bits utilisés pour coder les entiers suceptibles d'être affectés à la variable n.
- Dans certaines circonstances (variable d'instance), une valeur par défaut sera affecté à une telle variable lors de sa définition.

Les variables de type référence

- Toute variable dont le type associé n'est pas un type primitif est qualifiée de type **référence**.
- Rôle : référencer/repérer/adresser une zone mémoire contenant un objet du type correspondant.
- Leur déclaration ne donne lieu à aucune allocation en mémoire susceptible de contenir une grandeur du type référence. Seul l'espace nécessaire à la mémorisation d'une adresse est alloué.

Classes imbriquées

- Possibilité de déclarer une classe à l'intérieure d'une autre classe.
  - classe intérieure est appelée **classe imbriquée**
  - la classe imbriqué est membre de la classe qui la délimite
  - la classe imbriqué peut avoir des attributs d'accès

```
public class Externe
{
public class interne
{
// détails de la classe interne
}
// Plus de membres de la classe Externe }
```

- Externe est appelé **classe de niveau supérieure**

Classes imbriquées

- Classe Interne n'a de sens que dans le contexte d'un objet de type Externe
- La classe imbriquée doit avoir un lien particulier avec la classe qui la délimite
- Aucun objet de la classe imbriquée n'est créé, à moins, qu'ils ne soient conu par le constructeur de la classe qui l'a délimite

Externe exterieure = new Externe();
aucun objet de la classe imbriquée Interne n'est crée.
Externe.Interne interieur = new exterieure.new Interne(); // définir un objet de la classe imbriquée

Classes imbriquées

- Pour produire des objets d'un type de classe imbriquée indépendant des objets de la classe qui l'a délimite, vous pouvez déclarer la classe imbriqué **static**. Par exemple,

```
public class Externe
{
public static class Skinterne { //détails de Skinterne }
}
// classe imbriquée
public class Interne {détails de la classe interne }
// Plus de membres de la classe Externe
}
```

- Désormais, avec Skinterne à l'intérieure de Externe déclaré comme static, nous pouvons déclarer des objets de cette classe imbriquée, indépendamment des objets de Externe et sans tenir compte du fait que nous avons ou non créé des objets Externe. Par exemple :

```
Externe.Skinterne exemple = Externe.Skinterne();
```

Classes imbriquées locales

- Vous pouvez définir une classe à l'intérieur d'une méthode
  - **Classe imbriquée locale** ou **classe interne locale**
  - Vous ne pouvez créer des objets d'une classe interne locale que...localement
  - Pratique lorsque le calcul d'une méthode fait appel à une classe spécialisée qui n'est pas nécessaire ou qui est utilisée ailleurs.

Paquetages

Paquetages

- **Paquetage :** collection de classes nommées.
- regroupement des classes dans un paquetage a pour fonction de simplifier l'ajout de classe dans votre code.
- Les noms utilisés pour les classes n'interfèrent pas avec ceux des classes d'un autre paquetage, ou avec votre programme car ils sont qualifiés par le nom du paquetage.

Paquetages

- En java
  - Chaque classe est contenu dans un paquetage, y compris celle que nous avons définies dans nos exemples.
  - Jusqu'à présent pas de référence implicite car nous avons utilisé de manière implicite le **paquetage par défaut** pour centenir nos classes, et celui-ci n'a pas de nom.
    - Celui utilisé jusqu'à présent est `java.lang`
    - Il existe d'autres paquetages standards que vous devrez inclure explicitement lors de leur utilisation

Paquetages : Empaqueter vos classes

- Ajout de l'instruction de paquetage comme **première instruction** dans le fichier source qui contient la définition de la classe
  - (Mot Clé) **package** + **nom du paquetage** + **;**

```
package Geometrie;
public class Sphere
{
// détails de la définition de class
}
```

- Toute classe que vous voulez inclure dans le paquetage `Geometrie` doit contenir la même instruction de paquetage au début
- Attention : Vous devez enregistrer tous les fichiers des classes du paquetage dans un répértoire portant le même nom que le paquetage à savoir `Geometrie`

Paquetages et structure du répertoire

- Un paquetage est intimement lié à la structure du répertoire dans lequel il est stocké.
  - Une classe `NomClasse` ⇒ Fichier `NomClasse.java`
  - Tous les fichiers de classes au sein d'un paquetage, `NomPaquetage`, doivent être inclus dans un répértoire également appelé `NomPaquetage`.
  - Pouvez compiler la source d'une classe au sein d'un paquetage et faire générer le ficher `.class` dans un répertoire différent, mais le nom du répertoire doit toujours être le même que celui du paquetage.
  - Un paquetage ne doit pas nécessairement porter un seul nom. Vous pouvez spécifier un nom de paquetage par une suite de noms séparés par des points.

```
package Geometrie.Formes3D
```

```
package Geometrie.Formes2D
```

- ⇒ `Formes2D`, `Formes3D` sont des sous répértoires de `Geometrie`

Ajout de classes à un programme à partir d'un paquetages

- En supposant qu'ils ont été définies par le mot clé `public`
- Vous pouvez ajouter n'importe quelle classe d'un paquetage dans le code de votre programme au moyen d'instrcutions **import** en employant les noms des classes

```
import Geometrie.Formes3D.*; // Inclure toutes les classes du paquetage
```

- * sélectionne toutes les classes du paquetage $\Rightarrow$ faire référence à tous les classes publiques du paquetage
- Les noms des autres classes de votre programme doivent être différents de ceux des classes du paquetage.

```
import Geometrie.Formes3D.Sphere; // inclure la classe sphere
```

Paquetages et noms de programmes

- un paquetage crée un environnement propre pour nommer vos classes
  - $\Rightarrow$ la raison principale de l'existence de paquetages en java.
- Vous pouvez spécifier les noms des classes dans un paquetage sans vous préoccuper de savoir si les mêmes noms ont été utilisés ailleurs
- Java traite le nom du paquetage comme faisant partie du nom de la classe, comme un préfixe.
  - $\Rightarrow$ Le nom entier de la classe `Sphere` du paquetage Geometrie.Formes3D est en faite `Geometrie.Formes3D.Sphere`

Héritage

- Définir une nouvelle classe à partir d'une autre ⇒ Dérivation
  - Nouvelle classe : **classe dérivée**
  - Classe originale : **superclasse** ou classe de base.
- Définir une nouvelle classe en utilisant une autre en utilisant le mot **extends**

```
class Chien
Membres de la classe Chien...
```

```
class Epagneul extends Chien
Membres de la classe Epagneul....
```

- **Conséquence :**
  - Un objet de type Epagneul contiendra des membres hérités de la classe Chien + Les membres propres à Epagneul (Membres spécifiques caractérisants Epagneul).
  - Un objet Epagneul est un objet spécialisée d'un objet Chien.

Héritage

- Bonne modélisation de la vie réelle.
- La dérivation d'une nouvelle classe d'une classe de base est un processus additif en ce qui concerne la définition de classe.
  - Les membres supplémentaires définissent ce qui fait que l'objet de la classe dérivée est différent de celui de la classe de base.
  - Tous les membres déclarés dans la nouvelle classe sont ajoutés à ceux qui sont déjà membres de la classe de base.

Héritage

- à l'exception de la classe Object (qui na pas de superclasse) toute classe possède une superclasse.
- En absence d'une superclasse explicite, toute classe crée est automatiquement une sous-classe de Object.
- Une classe peut dériver d'une classe qui dérive d'une autre classe et ainsi de suite, jusqu'à la superclasse Object.
- Une telle classe est dite descendante des toutes les classes dans la chaine d'héritage.

Héritage

- Attention : une sous-classe hérite tous les membres de sa superclasse.
- Les constructeurs ne sont pas des membres !
- Donc, ils ne sont pas hérités par la sous-classe, mais ils peuvent être évoqués par la sous-classe.

L'arbre de la relation d'héritage

- La relation d'héritage forme un arbre sur les classes.
- Toute classe qui n'hérite pas explicitement d'une autre classe hérite de la classe `Object`.
- La classe `Object` est la racine de cet arbre.

L'arbre d'héritage

Soit une classe B qui hérite d'une classe A.

On dira que A est la classe mère, ou super-classe, et B `la classe fille, ou sous-classe`.

- Toute instance de B est une instance de A.
- Toute instance de B posséde tous les membres de A plus les membres dèfinis dans B.
- Au niveau de la classe, tous les membres statiques de A sont des membres statiques de B (et B posséde en plus les membres statiques dèfinis dans B).
- On peut redèfinir dans B les mèthodes de A.

L'accès aux membres d'une classe

- Cela n'implique pas que tous les membres de données définis dans la classe de base sont accessibles aux méthodes spécifiques de la classe dérivée. Certains le sont d'autres non.
- Un **membre hérité** d'une classe de base est **accessible** au sein de la classe dérivée.
- S'il n'est pas accessible alors il ne s'agit pas alors d'un membre hérité.
- Toutefois les membres de classe qui ne sont hérités font quand même partie de l'objet de la classe dérivée.

Membres "private" dans la superclasse

- La sous-classe n'a pas accès aux membres "private" de la superclasse.
- Mais les classes imbriquées de la superclasse ont accès aux membres private de la superclasse.
- Donc, les classes imbriquées public et protected dans la superclasse (auxquelles la sous-classe a accès) fournissent une possibilité daccès indirect aux membres private de la superclasse.

Membres "private" dans la superclasse

- Exemple

```
class A
{
private int x;
protected class AA
{
public int getX() { return x; }
}
public AA obj = new AA();
}
class B extends A {
public int getX() { return this.obj.getX(); }
}
class C {
public static void main(String[] args) {
System.out.println((new B()).getX());
}
}
```

L'accès aux membres d'une classe

Conclusion :

Une instance d'une sous-classe ne peut pas accéder directement aux membres privés de ses super-classes.

L'accès ne peut se faire que via des méthodes `public` ou `protected` (ou `package` suivant la localisation de la sous-classe).

Masquer des membres de données

- Membre de donnée peut avoir le même nom dans la classe de base et dérivée. (Pas recommandée)
- $\Rightarrow$ les membres de données de la classe de base sont hérités. mais masqués par les membres de la classe dérivée.
- Toute utilisation du nom du membre de la classe dérivée fera toujours référence au membre défini en tant que partie de la classe dérivée.
- Pour faire référence au membre de la classe héritée, vous devez la qualifier à l'aide du mot clé super.
  - Exemple : Supposons que vous posséder un membre valeur comme membre de la classe de base et un membre du meme nom dans la classe dérivée. valeur fera référence au membre de la classe dérivée et super.valeur au membre hérité de la classe de base.
- Attention : pas de possibilité d'utiliser super.super.quelqueschose.

Méthodes hérités

- Les méthodes ordinaires d'une classe de base (sauf constructeurs) sont héritées dans une classe dérivée (pareil que les de données d'une classe de base).
- Les méthodes déclarées comme `private` ne sont pas héritées. et celle que vous déclarer sans attribut d'accès ne sont pas héritées que si vous définissez la classe dérivée dans le même paquetage que la classe de base. Les autres méthodes sont toutes hérités.
- Les constructeurs sont différents des autres méthodes ordinaires. Les constructeurs d'une classe de base ne sont jamais hérités. quels que soient leurs attributs.

Objets d'une classe hérité

Faire un digramme

- Le constructeur de la classe de base peut être appelé dans la classe dérivée même s'ils sont pas hérités. Vous pouvez les appeler pour initialiser les membres de la classe de base.
- Même si vous n'appelez pas le constructeur de la classe de base à partir du constructeur de la classe dérivée, le compilateur essaiera de le faire pour vous.
- Puisqu'un objet de la classe dérivée englobe un objet de la classe de base, l'utilisation d'un constructeur de la classe de base constitue un bon moyen d'initialiser la partie de base d'un objet d'une classe dérivée.

Objets d'une classe hérité

```
public class Animal
{
public Animal(String unType)
{
type = new String(unType);
}
public String toString()
{
return "il s'agit d'un "+type;
}
private String type;
}
```

Objets d'une classe hérité

```
public class Chien extends Animal
{
public Chien(String unNom)
{
super("Chien"); //Appelle le constructeur de base
NomduChien = unNom;
RaceduChien = "Inconnu";
}
public Chien(String unNom, String uneRace)
{
super("Chien"); //Appelle le constructeur de base
NomduChien = unNom;
RaceduChien = uneRace;
}
private String NomduChien;
private String RaceduChien;
}
```

super est utilisé pour faire référence au constructeur de la classe de base.

Redéfinir une méthode d'une classe de base

- Vous pouvez redéfinir une méthode de la classe de base dans une classe dérivée, comportant la même signature q'une méthode de classe de base.
- L'attribut d'accès de la méthode dans une classe dérivée peut être identique à celui de la classe de base, ou moins restrictif, mais il ne peut pas être plus restrictif
  - $\Rightarrow$ Ex. Toute méthode est déclarée comme `public` dans une classe de base doit être également comme `public` dans la classe dérivée.
  - $\Rightarrow$ vous ne pouvez pas omettre l'attribut d'accès de la classe dérivée ou le spécifier comme privée ou protected.

La redéfinition de méthodes

Impératif : On doit respecter la signature de la méthode qu'on redéfinit!

- ses paramètres (nombre, type, ordre)
- son type de retour
- son attribut d'accessibilité : on peut élargir son accès.

La redéfinition de méthodes

Impératif : On doit respecter la signature de la méthode qu'on redéfinit!

- ses paramètres (nombre, type, ordre)
- son type de retour
- son attribut d'accessibilité : on peut élargir son accès.

Une méthode `package` peut être redéfinie en une méthode `public`.

Une méthode `public` ne peut pas devenir `private` dans une sous-classe.

Polymorphisme

- Le polymorphisme en Java est la capacité des sous-classes d'avoir leur propre comportement et, en même temps, de partager quelques fonctionnalités avec dautres sous-classes.
- Exemple :

```
class Vtt extends Velo {
private String suspension;
...
void imprimeEtat() {
System.out.println("vitesse : " + vitesse);
System.out.println("suspension : " + suspension);
}
}
```

Polymorphisme

```
class DemoVelo {
public static void main(String[] args) {
Velo velo1 = new Velo(20);
Velo velo2 = new Vtt(20, "dual");
velo1.imprimeEtat();
System.out.println();
velo2.imprimeEtat();
}
}
```

Résultat

```
vitesse : 20
vitesse : 20
suspension : dual
```

Polymorphisme

- La JVM évoque la méthode appropriée de l'objet référencié par chaque variable.
- Cela sappelle **virtual method invocation**, ou "late binding".

Classes et méthodes "final"

- Le modificateur **final** est utilisé pour indiquer qu'une méthode ne peut pas être redéfinie.
- Ils sont utiles dans le cas des méthodes appelées par les constructeurs d'une classe. Si une sous-classe redéfinie une telle méthode, cela peut avoir des conséquences indésirables.
- Une classe entière peut être déclarée comme "final". Cela veut dire que la classe ne peut pas avoir des sous-classes.

Graphe de la relation de typage : Conversions de type

supertype d'une classe A =
l'ensemble des classes "au-dessus" de `A` dans l'arborescence de la relation d'hèritage + l'ensemble des interfaces que `A` implèmente.

Toute instance de `A` est une instance de tous les èlèments de son supertype.

Pour les types objets comme pour les types primitifs, la conversion de type est possible si on respecte la règle d'èlargissement du domaine.

Conversions de type

Un objet **o** peut etre affectè a une variable d'un type **A** si **A** est le type de **o** ou fait partie du sur-type de **o**.

Toute instance d'`Etudiant` est une instance de `Personne`, mais la rèciproque est fausse.

Par contre, dans la suite du code, seuls les mèthodes et attributs accessibles de **A** pourront etre invoquès sur **o**.

Exemple

```
Personne etud = new Etudiant("Durand", "Paul",
"Licence Informatique", "Universite d'Artois"); // OK
System.out.println(etud.getFormation());
// refuse !!  erreur a la compilation!!
// getFormation() n'est pas une methode de Personne
Etudiant p = new Personne("Dupont","Jacques");
// !!  refuse // !!  erreur a la compilation
```

Le polymorphisme

Le polymorphisme :

a l'exècution, Java choisit la mèthode a interprèter en fonction du type de l'objet a qui est envoyè le message.

Exemple

```
// quelque part
public static void afficheToi(Personne p)
System.out.println(p);
// et plus loin...
Personne p1 = new Personne("Dupont","Jacques");
Personne p2 = new Etudiant("Durand", "Paul", "Licence
Informatique", "Universite d'Artois");
afficheToi(p1);
afficheToi(p2);
```

Exemple

provoquera l'affichage suivant :

```
Jacques Dupont
Paul Durand inscrit en Licence Informatique a
Universite d'Artois
```

Hèritage multiple?

Et si on veut crèer la classe des ètudiants-salariès?

```
public class EtudiantSalarie extends Etudiant,
Salarie  // !!  Refuse !!  Erreur a la compilation !!
```

Il n'y a pas d'héritage multiple en Java.
Une classe ne peut hériter que d'une seule classe.

Hèritage multiple?

L'hèritage multiple pose le problme du conflit de code : que faire lorsqu'on hèrite de deux classes qui proposent deux implèmentations diffèrentes d'une mme mèthode?

Java refuse donc l'hèritage multiple.

Les interfaces permettent de lever cette limitation, car une classe peut implèmenter plusieurs interfaces.

Privilègier l'hèritage sèmantique

Question

1. Un `Rectangle` est-il un `Carrè` dèfini avec un `cotè` supplèmentaire ?

   ou bien

2. Un `Carrè` est-il un `Rectangle` dont les deux cotès sont de la meme longueur ?

Privilègier l'hèritage sèmantique

Question

1. Un `Rectangle` est-il un `Carrè` dèfini avec un `cotè` supplèmentaire ?

   ou bien

2. Un `Carrè` est-il un `Rectangle` dont les deux cotès sont de la meme longueur ?

Rèponse : Un `Carrè` est un `Rectangle`!

Privilègier l'hèritage sèmantique

```
class Rectangle{
int longueur;
int largeur; Rectangle(int longueur, int largeur){
this.longueur = longueur;
this.largeur = largeur;
} int perimetre(){
return 2*(longueur+largeur);
}
int aire(){
return longueur * largeur;
}
}
```

Privilègier l'hèritage sèmantique

```
class Carre extends Rectangle{
Carre(int longueur){
super(longueur, longueur);
}
}
```

Ramasse-miettess (garbage collector)

- La plate-forme Java permet la création d'un nombre illimité d'objets.
- Par ailleurs, il n'est pas nécessaire de se préoccuper de leur suppression : la JVM supprime les objets qui ne sont plus utilisés.
- Un objet est éligible à la suppression une fois qu'il n' y a plus de référence à l'objet. C'est- à-dire, dès que :
  - toutes les variables qui contiennent la référence à l'objet cessent d'exister ; ou
  - toutes les variables qui contiennent la référence à l'objet sont affectées à la valeur null.

Accesseurs

Une bibliothèque gère plusieurs livres : on passe par un tableau.

```
public class Bibliotheque {
public static final int NB_MAX_LIVRES = 10000;
private Livre[] biblio;
private int dernier = -1; // indice du dernier livre
// mis dans la bibliothèque
private String nom;
public Bibliotheque(String unNom){
nom = unNom;
biblio = new Livre[NB_MAX_LIVRES];
}
// et une première méthode pour démarrer :
// affichage de tous les livres d'un auteur particulier
public void afficheLivresAuteurs(String auteur){
for(int i = 0; i ¡= dernier; i++){
if (biblio[i].getAuteur().equals(auteur))
System.out.println(biblio[i]);
}
}
}
```

Accesseurs

Ajoutons un nouvel accesseur dans `Bibliotheque`:

```
private Livre[] biblio;
public Livre[] getBiblio(){
return biblio;}
```

et maintenant . . .

Accesseurs

Ajoutons un nouvel accesseur dans `Bibliotheque`:

```
private Livre[] biblio;
public Livre[] getBiblio(){
return biblio;}
```

et maintenant ...

```
// ailleurs...
Bibliotheque laBu;
...
Livre[] bib = laBu.getBiblio();
for(int i=0;i¡bib.length;i++)
bib[i] = null;
```

ERREUR!!

Accesseurs

Conclusion : il faut se méfier des accesseurs en lecture et conserver cachées les propriétés qui ne concernent pas les autres. . .

Accesseurs

Conclusion : il faut se méfier des accesseurs en lecture et conserver cachées les propriétés qui ne concernent pas les autres...

Oui, mais alors

```
//class Personne
public String getNom(){
return nom;
}
```

est-ce dangereux ?

Non!

```
// class Test
Personne p = new Personne("Dumoulin","Isabelle",20);
String unNom = p.getNom();
unNom = unNom.toUpperCase();
// unNom contient "DUMOULIN"
// p.nom contient "Dumoulin"
```

Les instances de **String sont non modifiables** (comme les instances de classes enveloppes).

Forcer un type en Java

- Java langage fortement typé
  - le type de donnée est associé au nom de la variable, plutt qu'à sa valeur. (Avant de pouvoir être utilisée une variable doit être déclarée en associant un type à son identificateur).
- la compilation ou l'exécution peuvent détecter des erreurs de typage
- Dans certains cas, nécessaire de forcer le compilateur à considérer une expression comme étant d'un type qui n'est pas son type réel ou déclaré
- On utilise le cast ou transtypage: (type-forcé) expression
  - Exemple
  - int i = 64;
  - char c = (char)i;

Casts entre type primitifs

- Un cast entre types primitifs peut occasionner une perte de données
- Par exemple, la conversion d'un int vers un short peut donner un nombre complètement différent du nombre de départ.
  - **int i = 32768;**
  - **short s = (short) i;**
  - **System.out.println(s); -32767;**
- Un cast peut provoquer une simple perte de précision
  - Par exemple, la conversion d'un long vers un float peut faire perdre des chiffres significatifs mais pas l'ordre de grandeur
  - **long l1 = 928999999L;**
  - **float f = (float) l1;**
  - **System.out.println(f);** $\rightarrow$ **9.29E8**
  - **long l2 = (long) f;**
  - **System.out.println(l2);** $\rightarrow$ **929000000**

Casts entre types primitifs

- Les affectations entre types primitifs peuvent utiliser un cast implicite si elles ne peuvent provoquer qu'une perte de précision (ou, encore mieux, aucune perte)
  - int i = 130;
  - double x = 20 * i;
- Sinon, elles doivent comporter un cast explicite
  - short s = 65; // cas particulier affectation int "petit"
  - s = 1000000; // provoque une erreur de compilation
  - int i = 64;
  - byte b = (byte)(i + 2); // b = 66
  - char c = i; // caractère dont le code est 64 '@'
  - b = (byte)128; // b = -128 !

Casts entre entiers et charactères

- La correspondance **char** $\rightarrow$ **int**, long s'obtient par cast implicite
- Les correspondances **char** $\rightarrow$ **short, byte**, et **long**, **int**, **short** ou **byte** $\rightarrow$ **char** nécessitent un cast explicite.
  - int i = 80;
  - char c = 68; // caractère dont le code est 68
  - c = (char)i;
  - i = c;
  - short s = (short)i;

transtypage dans les affectations

- Lorsque le type du résultat d'une expression à droite et d'une instruction d'affectation diffère du type de la variable à gauche, un transtypage automatique est appliqué tant qu'il n'y a pas de risque de perte d'informations. Si vous prenez les types de base que nous avons vu jusqu'à présent dans l'ordre **byte** $\rightarrow$ **short** $\rightarrow$ **int** $\rightarrow$ **long** $\rightarrow$ **float** $\rightarrow$ **double**

Type booléen

Le type **boolean** a les deux valeurs **true** et **false**.

- c'est un type à part entière, indépendant des types numériques ou **char**;
- pas de conversion possible;
- en particulier, pas comme en C ou en C++, o
  - une expr. évaluée à zéro n'est pas équivalente à `false`
  - une expr. évaluée à une valeur différente de zéro n'est pas équivalente à `true`

  (différence avec le C et le C++)

Programmation Orientée Objet
Cours3

Mohamed Nabil Saidi

msaidi@insea.ac.ma

26 Mars 2012

Plan

- Type énumération
- Les Classes Enveloppes
- Conversion & Transtypage
- Polymorphisme

Type énumération

Type énumération

- Un type énuméré est considéré comme un genre particulier de classe
- Classe définie par énumération explicite de ses instances
- En Java on utilise le mot-clé **enum** :

```
enum Jour {
LUNDI, MARDI, MERCREDI, JEUDI,
VENDREDI, SAMEDI, DIMANCHE
}
```

Type énumération

```
class DemoEnum {
Jour jour;
public DemoEnum(Jour jour){
this.jour=jour;
}
```

```
public void agenda {
swith( jour){
case LUNDI: System.out.println(TP et TD.); break;
case MERCREDI: System.out.println(Cours.); break;
case VENDREDI: System.out.println(TP); break;
default: System.out.println(Rien a faire!!); break;
}
}
```

Type énumération

```
public static void main(String[] args) {
new DemoEnum(Jour.LUNDI).agenda();
new DemoEnum(Jour.MARDI).agenda();
new DemoEnum(Jour.MERCREDI).agenda();
new DemoEnum(Jour.JEUDI).agenda();
new DemoEnum(Jour.VENDREDI).agenda();
new DemoEnum(Jour.SAMEDI).agenda();
new DemoEnum(Jour.DIMANCHE).agenda();
}
}
```

Type énumération

- Les éléments énumérés (comme SAMEDI) ne sont rien d'autres que des instances de la classe définie
- Il n'est pas possible de construire ultérieurement d'autres instances que celles énumérées immédiatement dans la définition de la classe
- **Un type énumération est une classe**. Son corps de déclaration peut contenir des méthodes et d'autres attributs.
- Le compilateur crée par défaut la méthode values() qui retourne un tableau contenant les valeurs de l'énumération dans lordre qu'ils sont déclarés :

```
Jour [] semaine = Jour.values();
```

La commande "for-each"

- Cette commande peut etre utilisé avec les tableaux (donc, aussi avec les énumérations) :
- **for (Type Identificateur : Expression) Commande**.

Exemple :

```
for ( int item : {1,2,3,4,5 } ) {
System.out.println("Numero courant :" + item);
}
```

équivalent a :

```
int []tableau={1,2,3,4,5 } ;
for ( int i=0;i< tableau.length;i++ ) {
int item=tableau[i];
System.out.println("Numero courant :" + item);
}
```

La commande "for-each"

```
for ( Jour j : Jour.values() ) {
new DemoEnum(j).agenda();
}
```

équivalent a :

```
Jour []tableau=Jour.values() ;
for ( int i=0;i< tableau.length;i++ ) {
Jour j=tableau[i];
new DemoEnum(j).agenda();
}
```

Exercice : Jeu de cartes

- Déclarez une classe Carte avec deux attributs immuables, rang et couleur, un constructeur, des accesseurs et une méthode toString().
- Déclarez une classe Jeu qui contient un tableau de cartes. Déclarez son constructeur et une méthode imprimeEtat().
- Déclarez une classe DemoJeu qui instancie un jeu et imprime son état.

La commande "for-each" avec un type

Reponse

```
enum Rang {
Deux, Trois, Quatre, Cinq, Six,
Sept, Huit, Neuf, Dix,
Valet, Dame, Roi, As
}
enum Couleur {
Carreau, Trefle, Coeur, Pique
}
```

La commande "for-each" avec un type

```
class Carte {
private final Rang rang;
private final Couleur couleur;
public Carte(Rang rang, Couleur couleur){
this.rang = rang;
this.couleur = couleur;
}
public Couleur getCouleur() {
return couleur;
}
public Rang getRang() {
return rang;
}
public String toString() {
return rang + " de " + couleur;
}
}
```

La commande "for-each" avec un type

```
class Jeu {
private static Carte[] cartes = new Carte[52];
public Jeu(){
int i = 0;
for (Couleur couleur : Couleur.values()) {
for (Rang rang : Rang.values()) {
cartes[i++] = new Carte(rang, couleur);
}
}
}
public void imprimeEtat() {
for (Carte carte : cartes)
System.out.println(carte.toString());
}
}
```

La commande "for-each" avec un type

```
class DemoJeu {
public static void main(String[] args){ Jeu jeu = new Jeu();
jeu.imprimeEtat(); }
}
```

Les Classes Enveloppes

Les Classes Enveloppes

Il existe un type objet correspondant à chaque type primitif:

- `Character`
- `Boolean`
- `Integer`
- `Float`
- `Double`
- `Long`
- `Short`
- `Byte`

Description des classes enveloppes

Elles contiennent toutes :

- un constructeur prenant en argument une valeur du type primitif correspondant
- un constructeur avec une **String** en argument
- et une fonction qui retourne la valeur de l'objet dans le type primitif correspondant

Les instances de ces classes sont **non-modifiables**.

Passage type primitif - instance d'une classe enveloppe

Jusqu'au JDK 1.4

```
Integer[] tabInt = new Integer[3];
tabInt[0] = new Integer(5);
int i = tabInt[0].intValue();
```

Passage type primitif - instance d'une classe enveloppe

À partir du JDK 1.5, c'est le compilateur qui fait le travail!
L'empaquetage/dépaquetage est fait automatiquement.

```
Integer[] tabInt = new Integer[3];
tabInt[0] = 5;
int i = tabInt[0];
```

Les instances de Integer sont non-modifiables

Jusqu'au JDK 1.4

```
Integer i = new Integer(3);
// pour incrémenter i
i = new Integer(i.intValue()+1);
```

Les instances de Integer sont non-modifiables

Avec le JDK 1.5, la syntaxe est plus légère (mais l'implémentation est la même!)

```
Integer i = new Integer(3);
// pour incrémenter i
i = i+1;
```

C'est le compilateur qui effectue la conversion vers **int** et la création d'un nouvel **Integer**.

Les instances de Integer sont non-modifiables

Lorsqu'on a des calculs à faire, il faut les faire sur les types primitifs!

```
public Integer beaucoupDeTrucsACalculer(Integer i){
// conversion dans un type primitif
int petitI = i;
// tous les calculs se font maintenant sur petitI
...
// et la conversion dans l'autre sens sera faite automatiquement
return petitI;
}
```

Conversion & Transtypage

Forcer un type en Java

- Java langage fortement typé
  - le type de donnée est associé au nom de la variable, plutt qu'à sa valeur. (Avant de pouvoir être utilisée une variable doit être déclarée en associant un type à son identificateur).
- la compilation ou l'exécution peuvent détecter des erreurs de typage
- Dans certains cas, nécessaire de forcer le compilateur à considérer une expression comme étant d'un type qui n'est pas son type réel ou déclaré
- On utilise le cast ou transtypage: (type-forcé) expression
  - Exemple
  - int i = 64;
  - char c = (char)i;

Casts entre type primitifs

- Un cast entre types primitifs peut occasionner une perte de données
- Par exemple, la conversion d'un int vers un short peut donner un nombre complètement différent du nombre de départ.
  - **int i = 32768;**
  - **short s = (short) i;**
  - **System.out.println(s); -32767;**
- Un cast peut provoquer une simple perte de précision
  - Par exemple, la conversion d'un long vers un float peut faire perdre des chiffres significatifs mais pas l'ordre de grandeur
  - **long l1 = 928999999L;**
  - **float f = (float) l1;**
  - **System.out.println(f);** $\rightarrow$ **9.29E8**
  - **long l2 = (long) f;**
  - **System.out.println(l2);** $\rightarrow$ **929000000**

Casts entre entiers et charactères

- La correspondance **char** $\rightarrow$ **int, long** s'obtient par cast implicite
- Les correspondances **char** $\rightarrow$ **short, byte**, et **long**, **int**, **short** ou **byte** $\rightarrow$ **char** nécessitent un cast explicite.
  - int i = 80;
  - char c = 68; // caractère dont le code est 68
  - c = (char)i;
  - i = c;
  - short s = (short)i;

transtypage dans les affectations

- Lorsque le type du résultat d'une expression à droite et d'une instruction d'affectation diffère du type de la variable à gauche, un transtypage automatique est appliqué tant qu'il n'y a pas de risque de perte d'informations. Si vous prenez les types de base que nous avons vu jusqu'à présent dans l'ordre **byte** $\rightarrow$ **short** $\rightarrow$ **int** $\rightarrow$ **long** $\rightarrow$ **float** $\rightarrow$ **double**

Type booléen

Le type **boolean** a les deux valeurs **true** et **false**.

- c'est un type à part entière, indépendant des types numériques ou **char**;
- pas de conversion possible;
- en particulier, pas comme en C ou en C++, ou :
  - une expr. évaluée à zéro n'est pas équivalente à `false`
  - une expr. évaluée à une valeur différente de zéro n'est pas équivalente à `true`

  (différence avec le C et le C++)

Surclassement

- La réutilisation du code est un aspect important de l'héritage, mais ce n'est peut être pas le plus important
- Le deuxième point **fondamental** est la relation qui relie une classe à sa superclasse :

  Une classe B qui hérite de la classe A peut être vue comme un sous-type (sous ensemble) du type défini par la classe A.

```
class Etudiant
{...}
```

```
class EtudiantSportif extends Etudiant {...}
```

Un *EtudiantSportif* **est un** *Etudiant*

L'ensemble des étudiants sportifs est inclus dans l'ensemble des étudiants

Surclassement

- tout objet instance de la classe B peut être aussi vu comme une instance de la classe A.
- Cette relation est directement supportée par le langage JAVA :
  - à une référence déclarée de type A il est possible d'affecter une valeur qui est une référence vers un objet de type B (**surclassement** ou **upcasting**)

```
Etudiant e;
e = new EtudiantSportif(...);
```

F hérite de E
D et E héritent de C
B et C héritent de A

```
C c;
c = new D();
c = new E();
c = new F();
c = new A(); //Erreur
c = new B(); //Erreur
```

Surclassement

- Lorsqu'un objet est "sur-classé" il est vu par le compilateur comme un objet du type de la référence utilisée pour le désigner
  - Ses fonctionnalités sont alors restreintes à celles proposées par la classe du type de la référence

```
class Etudiant {
String nom;
String prénom;
int age;
public Etudiant(String n, String p,int a
...)
public void affiche()
public int nbInscriptions()
}
class EtudiantSportif {..
String sportPratiqué;
public EtudiantSportif (String n, String
p,
int a, , String s, ...)
public void affiche()
public double bonusSportif()
}
```

```
EtudiantSportif es;
es = new
EtudiantSportif("DUPONT","Jean",
25,..,"Badminton",..);
Etudiant e;
e = es; // upcasting
e.affiche();
es.affiche();
e.nbInscriptions();
es.nbInscriptions();
es.bonusSportif();
e.bonusSportif(); // Le compilateur
refuse ce message : pas de méthode
bonusSportif définie dans la classe
Etudiant
```

Lien dynamique : Résolution de messages

- Que va donner e.affiche() ?

Etudiant e = new EtudiantSportif(
"DUPONT","Jean",25,..,"Badminton",..);

```
class Etudiant {
public void affiche(){
System.out.println(
"Nom : "+nom+"n" "Prénom : "+prénom+"n" "Age : "+age+ ...);
}
}
```

```
class EtudiantSportif extends Etudiant
{
public void affiche(){
super.affiche();
System.out.println( "Sport" : "+sport+"n" + ...);
}
}
```

Lien dynamique : Résolution de message

Lorsqu'une méthode d'un objet est accédée au travers d'une référence "surclassée",c'est la méthode telle qu'elle est définie au niveau de la classe effective de l'objet qui est en fait invoquée et exécutée

- Etudiant e = new EtudiantSportif("DUPONT","Jean",25,..,"Badminton",..);

Nom : DUPONT
Prénom : Jean
Age : 25
...
Sport : Badminton

Lien dynamique : Mécanisme de résolution de message

- Les messages sont résolus à l'exécution : la méthode exécutée est déterminée à l'exécution (run-time) et non pas à la compilation
  - à cet instant le type exact de l'objet qui recoit le message est connu
    - la méthode définie pour le type réel de l'objet recevant le message est appelée (et non pas celle définie pour son type déclaré).

```
public class A {
public void m() {
System.out.println("m de A");}
}
-------------------------
public class B extends A {
public void m() {System.out.println("m
de B");}
}
-------------------------
public class C extends B {}
```

A obj = new C();
obj.m();
⇒ **m de B**

- ce mécanisme est désigné sous le terme de **lien-dynamique** (dynamic binding, late-binding ou run-time binding)

Lien dynamique : Vérifications statiques

- **A la compilation :** seules des **vérifications statiques** qui se basent sur le type déclaré de l'objet (de la référence) sont effectuées
  - la classe déclarée de l'objet recevant le message doit posséder une méthode dont la signature correspond à la méthode appelée.

```
public class A {
public void m1()
{System.out.println("m1 de A");}
}
```

```
public class B extends A {
public void m1()
{System.out.println("m1 de B");}
public void m2()
{System.out.println("m2 de B");}
}
```

```
A obj = new B();
obj.m1();
obj.m2(); //Erreur
```

```
Test.java:21: cannot resolve symbol
symbol : method m2 ()
location: class A
obj.m2(); 1 error
```

- vérification statique : garantit dès la compilation que les messages pourront être résolus au moment de l'exécution

Choix de méthode, sélection du code

```
public class A {
public void m1() { System.out.println("m1 de A");}
public void m1(int x) {System.out.println("m1(x) de A");}
}
public class B extends A {
public void m1() {System.out.println("m1 de B");}
public void m2() {System.out.println("m2 de B");}
}
```

Le choix de la méthode à exécuter est effectué statiquement à la compilation en fonction du type des paramètres

```
invokevirtual  <Method m1() > // refaA.m1();
invokevirtual  <Method m1(int) > // refaA.m1(10);
invokevirtual  <Method m1() > //refA.m1();
```

```
A refA = new A();
refaA.m1();
refA.m1(10);
refA = new B();
refA.m1();
```

La sélection du code à exécuter est effectué dynamiquement à l'exécution en fonction du type effectif du récepteur du message

Polymorphisme

A quoi servent l'upcasting et le lien dynamique ?
A la mise en oeuvre du polymorphisme

- Le terme polymorphisme décrit la caractéristique d'un élément qui peut se présenter sous différentes formes.
- En programmation Objet, on appelle polymorphisme
  - le fait qu'un objet d'une classe puisse être manipulé comme s'il appartenait à une autre classe.
  - le fait que la même opération puisse se comporter différemment sur différentes classes de la hiérarchie.
- "Le polymorphisme constitue la troisième caractéristique essentielle d'un langage orienté objet après l'abstraction des données (encapsulation) et l'héritage" *Bruce Eckel "Thinking in JAVA"*

Polymorphisme

- En utilisant le polymorphisme en association à la liaison dynamique
  - plus besoin de distinguer différents cas en fonction de la classe des objets
  - possible de définir de nouvelles fonctionnalités en héritant de nouveaux types de données à partir d'une classe de base commune sans avoir besoin de modifier le code qui manipule l'interface de la classe de base
- Développement **plus rapide**
- Plus grande **simplicité** et **meilleure organisation** du code
- Programmes plus facilement **extensibles**
- Maintenance du code **plus aisée**

Downcasting

ClasseX obj;
ClasseA a = (ClasseA) obj;

- Le downcasting permet de forcer un type à la compilation
  - C'est une promesse que l'on fait au moment de la compilation.
- Pour que le transtypage soit valide, il faut qu'à l'exécution le type effectif de obj soit compatible avec le type ClasseA
  - Compatible : la même classe ou n'importe quelle sous classe de ClasseA (obj instanceof ClasseA)
- Si la promesse n'est pas tenue une erreur d'exécution se produit.
  - ClassCastException est levée et arrêt de l'exécution

java.lang.ClassCastException: ClasseX at Test.main(Test.java:52)

Upcasting/Downcasting

```
class A {
public void ma() {System.out.println("methode ma définie dans A");}
}
```

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

```
class C extends A { ... }
```

```
class D extends B { ... }
```

```
class E extends C {
public void ma() {System.out.println("methode ma redéfinie dans E");}
public void me() {System.out.println("methode me définie dans E");}
}
```

```
class F extends E {
public void mf() {System.out.println("methode mf définie dans f");}
public void me() {System.out.println("methode me redéfinie dans F");}
}
```

```
C c = new F();
```

Upcasting/Downcasting

	compilation	**exécution**
c.ma();	La classe C hérite d'une méthode ma	méthode ma définie dans E
c.mf();	Cannot find symbol : metod mf() Pas de méthode mf() définie au niveau de la classe C	
B b = c;	Incompatible types Un C n'est pas un B	
E e = c;	Incompatible types Un C n'est pas forcément un E	
E e = (E)c; e.me();	Transtypage (Dowcasting), le compilateur ne fait pas de vérification La classe E définit bien une méthode me	méthode me définie dans F
D d = (D) c;	Transtypage (Dowcasting), le compilateur ne fait pas de vérification	ClassCastException Un F n'est pas un D

Programmation Orientée Objet
Cours 4

Mohamed Nabil Saidi

msaidi@insea.ac.ma

29 Mars 2012

Plan

- Classes abstraites
- Interfaces

Classes abstraites

Classes abstraites

- **Utilité :**
  - Définir des concepts incomplets qui devront être implémentés dans les sous classes
  - Factoriser le code

```
public abstract class lesFrome {
protected double x, y;
public void deplacer(double dx, double dy)
{
x += dx; y+= dy;
}
public abstract double perimetre();
public abstract double surface();
}
```

Classes abstraites

- **classe abstraite** : classe non instanciable, c'est à dire qu'elle n'admet pas d'instances directes.
  - *Impossible de faire* **new ClassAbstraite(...)**
- **opération abstraite** : opération n'admettant pas d'implémentation au niveau de la classe dans laquelle elle est déclarée, on ne peut pas dire comment la réaliser.
- Une classe pour laquelle au moins une opération abstraite est déclarée est une classe abstraite (l'inverse n'est pas vrai).
- Les opérations abstraites sont particulièrement utiles pour mettre en oeuvre le polymorphisme.

Classes abstraites

- Une classe abstraite est une description d'objets destinée à être héritée par des classes plus spécialisées.
- Pour être utile, une classe abstraite doit admettre des classes descendantes **concrètes**.
- Toute classe **concrète** sous-classe d'une classe abstraite doit concrétiser toutes les opérations abstraites de cette dernière.
- Une classe abstraite permet de regrouper certaines caractéristiques communes à ses sous-classes et définit un comportement minimal commun.
- La factorisation optimale des propriétés communes à plusieurs classes par généralisation nécessite le plus souvent l'utilisation de classes abstraites.

Interfaces

Interfaces

- Dans certaines situations, il est important que différents groupes de programmeurs soient d'accord sur un contrat qui décrit comment leurs programmes interagissent.
- Chaque groupe doit être capable d'écrire son code sans avoir les détails du code des autres groupes.
- Les interfaces de Java servent à cet objectif.

Déclaration d'une interface

- Une interface est une collection d'opérations utilisée pour spécifier un service offert par une classe.
- Une interface peut être vue comme une classe abstraite sans attributs et dont toutes les opérations sont abstraites.

Dessinable.java

```
import java.awt.*;
public interface Dessinable {
public void dessiner(Graphics g);
void effacer(Graphics g);
}
```

- Toutes les méthodes sont abstraites
- Elles sont implicitement publiques
- Possibilité de définir des attributs à condition qu'il s'agisse d 'attributs de type primitif
- Ces attributs sont implicitement déclarés comme static final

Réalisation d'une interface

- Une interface est destinée à être réalisée (implémentée) par d'autres classes (celles-ci en héritent toutes les descriptions et concrétisent les opérations abstraites).
  - Les classes réalisantes s'engagent à fournir le service spécifié par l'interface

Déclaration d'une interface

- Composants (dans l'ordre) :
  1. Modificateurs (optionnels)
  2. Mot-clé interface' suivi du nom de l'interface (obligatoire)
  3. Mot-clé extends' suivi d'une liste de noms d'interfaces (optionnel)
  4. Corps de déclaration entouré par { et }

```
public interface InterfRegroupee extends Interf1,
Interf2 {
double E = 2.718282; // base des logarithmes
void faitQqChose (int i, double x);
int faitAutreChose (String s);
}
```

Réalisation d'une interface

- De la même manière qu'une classe étend sa super-classe elle peut de manière optionnelle implémenter une ou plusieurs interfaces
  - dans la définition de la classe, après la clause extends nomSuperClasse, faire apparatre explicitement le mot clé implements suivi du nom de l'interface implémentée

```
class RectangleDessinable extends Rectangle implements Dessinable {
public void dessiner(Graphics g){
g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
}
public void effacer(Graphics g){
g.clearRect((int) x, (int) y, (int)largeur, (int) hauteur);
}

}
```

- si la classe est une classe concrète elle doit fournir une implémentation (un corps) à chacune des méthodes abstraites définies dans l'interface (qui doivent être déclarées publiques)

Réalisation d'une interface

- Une classe JAVA peut implémenter simultanément plusieurs interfaces
- Pour cela la liste des noms des interfaces à implémenter séparés par des virgules doit suivre le mot clé implements

```
class RectangleDessinable extends Rectangle implements Dessinable,
Comparable {
public void dessiner(Graphics g){
g.drawRect((int) x, (int) y, (int) largeur, (int) hauteur);
}
public void effacer(Graphics g){ // Méthode de l'interface Dessinable
g.clearRect((int) x, (int) y, (int)largeur, (int) hauteur);
}
public int compareTo(Object o) { // Méthode de l'interface Comparable
if (o instanceof Rectangle)
...
} }
```

Interfaces

- En Java, une interface est un type (similaire à une classe) qui contient de constantes, les signatures et la valeur de retour des méthodes (mais pas leur corps de déclaration), et les classes imbriquées.
- Interfaces ne peuvent pas être instanciées. En revanche, elles peuvent être étendues et surtout implémentées.
- Un champ d'une classe peut être de type Interface
- Une variable dans une méthode peut être de type Interface

Déclaration d'une interface

- Une interface peut étendre plus d'une interface (une sorte d'héritage multiple).
- Toutes les méthodes d'une interface sont implicitement déclarées public (donc, le modificateur public peut être omis).
- Une interface peut contenir de déclarations de constantes. Elles sont implicitement déclarées
- public static final (donc, ces modificateurs peuvent être omis.)

Déclaration d'une interface

- **Exemple**

```
public interface Comparable {
// this et other doivent être des instances
// de la meme classe
// retourne 1, 0, -1 si this est plus grand,
// egal, ou plus petit que other
public int compareTo(Object other);
}
```

- Toute classe peut implémenter Comparable, si il y a une facon de comparer la ses objets.
- Si nous savons que la classe implémente Comparable, alors nous savons que nous pouvons comparer ses objets.

Héritage d'interfaces

- De la même manière qu'une classe peut avoir des sous-classes, une interface peut avoir des "sous-interfaces"
- Une sous interface
  - hérite de toutes les méthodes abstraites et des constantes de sa "superinterface"
  - peut définir de nouvelles constantes et méthodes abstraites

```
interface Set extends Collection
{
...
}
```

- Une classe qui implémente une interface doit implémenter toutes les méthodes abstraites définies dans l'interface et dans les interfaces dont elle hérite.

Implémentation d'une interface

```
class Rectangle implements Comparable {
public int largeur = 0;
public int hauteur = 0;
...
public int getSurface() { return largeur * hauteur; }
public int compareTo(Object other) {
Rectangle autreRect = (Rectangle)other;
if (this.getSurface() < autreRect.getSurface())
return -1;
else if (this.getSurface() > autreRect.getSurface())
return 1;
else
return 0;
}
}
```

Exercice

- 1. Déclarez une classe StringInverse qui contient une string inversée en implémentant l'interface suivante :

```
public interface CharSequence {
char charAt(int index);
int length();
CharSequence subSequence(int debut, int fin);
String toString();
}
```

Réponse

```
public class StringInverse implements CharSequence {
private String chaine;
public StringInverse(String str) {
char[] tab = new char[str.length()];
int j = str.length() - 1;
for (int i = 0; i < tab.length; i++)
tab[i] = str.charAt(j--);
chaine = new String(tab);
}
```

Réponse

```
public char charAt(int index) {
return chaine.charAt(index);
}
public int length() {
return chaine.length();
}
public CharSequence subSequence(int debut, int fin) {
return (CharSequence)chaine.subSequence(debut, fin);
}
public String toString() {
return chaine;
}
}
```

Programmation Orienté Objet
Cours 6

Mohamed Nabil Saidi

msaidi@insea.ac.ma

16 Avril 2012

Les Exceptions

Génération et gestion d'exception

1. Les exceptions servent à gérer les erreurs qui peuvent survenir durant l'exécution d'un programme
2. Les exceptions sont utilisés pour repérer les parties "dangereuses / à risque" dans un programme.
3. Lorsqu'une exception survient :
   - un objet représentant cette exception est créé;
   - cet objet est **jeté (thrown)** dans la méthode ayant provoqué l'erreur.
4. Cette méthode peut choisir :
   - de gérer l'exception elle-même,
   - de la passer sans la gérer.

De toutes façons, l'exception est captée (caught) et traitée en dernier recours par l'environnement d'exécution java.

Génération et gestion d'exception (2)

1. Les exceptions peuvent être générées :
   - par l'environnement d'exécution java
   - manuellement par du code
2. Les exceptions jetées (ou levées) par l'environnement d'exécution résultent de violations des règles du langage ou des contraintes de cet environnement d'exécution.

Structure générale du traitement des exceptions

```
try {
//bloc de code a surveiller
//peut lever une ou plusieurs exceptions
}
catch (ExceptionType1 exceptObj) {
//Traitement de l'exception du type1
}
catch (ExceptionType2 exceptObj) {
//Traitement de l'exception du type2
}
finally {
//code a exécuter avant de sortir (avec ou sans le traitement des
exceptions)
}
```

Exception non gerée

1. Considérons le code suivant ou une division par zéro n'est pas gérée par la programme :

ExcepDiv0.java

```
class ExcepDiv0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d; } }
```

2. Lorsque l'environnement d'exécution essaie d'exécuter la division, il construit un nouvel objet exception afin d'arrêter le code et de gérer cette condition d'erreur.
3. L'environnement d'exécution affiche la valeur en String de l'exception et la trace de la pile d'appels :

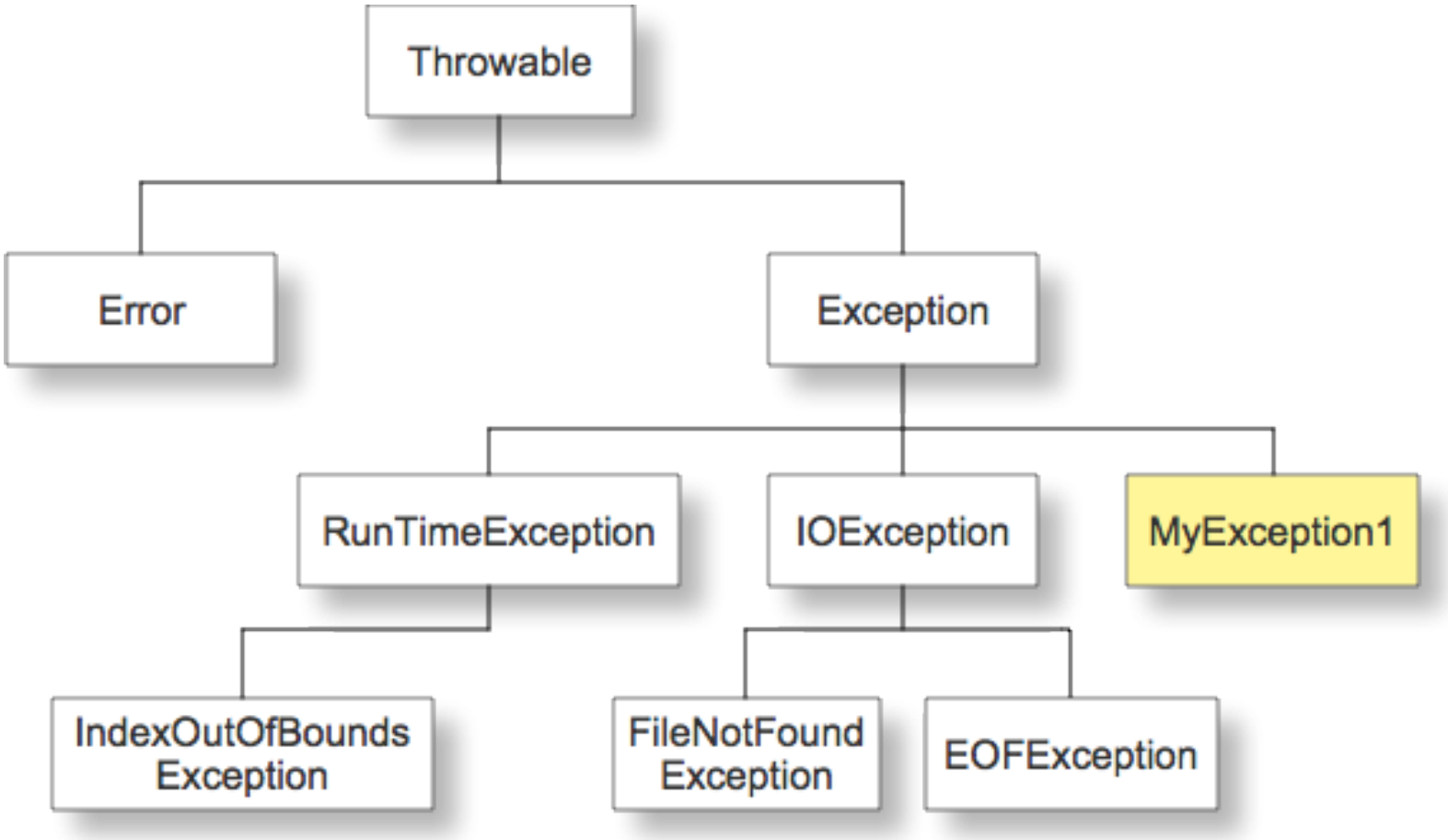
```
> java ExcepDiv0
java.lang.ArithmeticException: / by zero
at ExcepDiv0.main(ExcepDiv0.java:4)
```

Types d'exceptions

1. Une classe est au sommet de la hiérarchie des exceptions : **Throwable**
2. Deux sous-classes de Throwable :
   - Exception : conditions exceptionnelles que les programmes utilisateur devraient traiter.
   - Error : exceptions catastrophiques que normalement seul l'environnement d'exécution devrait gérer.
3. Une sous-classe d'Exception, RuntimeException, pour les exceptions de l'environnement d'exécution.

Types d'exceptions (2)

- En java les exceptions sont des objets
- toute exception doit être une instance d'une sous-classe de la classe java.lang.Throwable



Types d'exceptions (3)

```
Exception
    ClassNotFoundException
    CloneNotSupportedException
    IllegalAccessException
    InstantiationException
    InterruptedException
    NoSuchFieldException
    NoSuchMethodException
    RuntimeException
        ArithmeticException
        ArrayStoreException
        ClassCastException
        IllegalArgumentException
            IllegalThreadStateException
            NumberFormatException
        IllegalMonitorStateException
        IllegalStateException
        IndexOutOfBoundsException
            ArrayIndexOutOfBoundsException
            StringIndexOutOfBoundsException
        NegativeArraySizeException
        NullPointerException
        SecurityException
```

Instructions try et catch

1. Un bloc try est destiné a être protégé, gardé contre toute exception susceptible de survenir.
2. **try**{ . . . } délimite un ensemble d'instructions susceptibles de déclencher une(des) exception(s) pour la(les)quelles une gestion est mise en oeuvre
3. Juste après un bloc try, il faut mettre un bloc **catch** qui sert de gestionnaire d'exception. Le paramètre de l'instruction catch indique le type et le nom de l'instance de l'exception : **catch(TypeDexception e)** { . . . }
4. Cela permet d'intercepter ("attraper") les exceptions dont le type est spécifié et d'exécuter alors du code spécifique

Instructions try et catch (2)

ExcepDiv0.java

```
class ExcepDiv0 {
public static void main(String args[]) {
try {
int d = 0;
int a = 42 / d; }
catch (ArithmeticException e) {
System.out.println("Div par zero"); }
}
}
```

Instructions catch multiples

1. On peut gérer plusieurs exceptions a la suite l'une de l'autre.
2. Lorsqu'une exception survient, l'environnement d'exécution inspecte les instructions catch les unes après les autres, dans l'ordre ou elles ont été écrites.
3. Il faut donc mettre les exceptions les plus spécifiques d'abord.

Instructions catch multiples (2)

Excepmultiple.java

```
class Excepmultiple {
public static void main(String args[]) {
try {
String s = args[0];
int a = Integer.parsInt(" 123 ");
int b = Integer.parsInt( s );
System.out.println( a/b );
}
 catch (ArrayIndexOutOfBoundsException ex0)  {
//Traitement de l'exception en cas de dépassement de tableau
}
```

Instructions catch multiples (3)

Excepmultiple.java

```
catch (NumberFormatException ex1) {
//Traitement de l'exception en cas d'erreur de parsing
}
catch (Exception ex2) {
//Traitement des exceptions générales (telle que la division par 0)
}
}
}
```

Instructions catch multiples (4)

1. S'il n'y a pas de block catch alors le block finally est requis, sinon il est optionnel.
2. On peut attraper plusieurs exceptions dans un même block try, chaque classe d'exception a son propre traitement.
3. Les instructions d'un bloc try situées après une levée d'exception ne sont pas exécutées.
4. L'ordre des blocks catch est très important : Les classes les plus générales d'exception sont placées en dernier.

Instruction finally

- Les clauses catch sont suivies de manière optionnelle par un bloc finally qui contient du code qui sera exécuté quelle que soit la manière dont le bloc try a été quitté
- Le bloc finally permet de spécifier du code dont l'exécution est garantie quoi qu'il arrive :
  - le bloc try s'exécute normalement sans qu'aucune exception ne soit levée
  - le bloc try l'eve une exception attrapée par l'un des blocs catch.
  - le bloc try lève une exception qui n'est attrapée par aucun des blocs catch qui le suivent.

Instruction finally (2)

- Intérêt double :
  - permet de rassembler dans un seul bloc un ensemble d'instructions qui autrement auraient du être dupliquées
  - permet d'effectuer des traitements après le bloc try, même si une exception a été levée et non attrapée par les blocs catch

```
...
try {
// ouvrir un fichier
// effectuer des traitements
// susceptibles de lever une exception
// fermer le fichier
}
catch (CertaineException ex1){
// traiter l'exception
// fermer le fichier
}
catch (AutreTypeException ex2){
// traiter l'exception
// fermer le fichier
}
```

```
...
try {
// ouvrir un fichier
// effectuer des traitements
// susceptibles de lever une exception
}
catch (CertaineException ex1){
// traiter l'exception
}
catch (AutreTypeException ex2){
// traiter l'exception
}
finally {
// fermer le fichier
}
```

Instruction throw

1. Elle permet de générer une exception, via un appel de la forme :
   trow ThrowableInstance ;
2. Cette instance peut être crée par un new ou être une instance d'une exception déjà existante (sous-classe de Throwable).
3. Le flux d'exécution est alors stoppé et le bloc try immédiatement englobant est inspecté, afin de voir s'il possède une instruction catch correspondante à l'instance générée.
4. Si ce n'est pas le cas, le $2^{\text{ième}}$ bloc try englobant est inspecté ; et ainsi de suite.

Instruction throw (2)

ThrowDemo.java

```
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
}
catch (NullPointerException e2) {
System.out.print("attrapee ds demoproc()");
throw e2;
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e1) {
System.out.print("attrapee ds main()");
}
}
```

Instruction throws

1. Si une méthode est susceptible de générer une exception qu'elle ne gère pas, elle doit le spécifier, de façon que ceux qui l'appellent puissent se prémunir contre l'exception.
2. L'instruction throws est utilisée pour spécifier la liste des exceptions qu'une méthode est susceptible de générer.
3. Pour la plupart des sous-classes d'Exception, le compilateur forcera à déclarer quels types d'exception peuvent être générées (sinon, le programme ne compile pas).
4. Cette règle ne s'applique pas à Error, RuntimeException ou à leurs sous-classes.

Instruction throws (2)

L'exemple suivant ne se compilera pas :

ThrowsDemo1.java

```
class ThrowsDemo1 {
static void proc() {
System.out.println("dans proc()");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
proc();
}
}
```

Ce programme ne se compilera pas parce que :

- proc() doit déclarer qu'elle peut générer IllegalAccessException
- main() doit avoir un bloc try/catch pour gérer l'exception en question.

Instruction throws (3)

L'exemple correct est :

ThrowsDemo1.java

```
class ThrowsDemo1 {
static void proc() throws IllegalAccessException {
System.out.println("dans proc()");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
proc();
 }
catch(IllegalAccessException e) {
System.out.println(e + "attrapee");
 }
}
}
```

Définition d'une classe d'exception

On crée une exception comme n'importe quelle autre classe :

ExcptDiv0.java

```
class ExcptDiv0 extends Exception {
public ExcptDiv0 (String s) {
super(s);
}
}
```

...

```
public void divise(double x, double y) throws ExceptDiv0 {
if (y < 0)
    throw new ExceptDiv0("Atention !! division par zero ");
return x/y;
}
```

Le programme qui utilisera cette méthode doit gérer l'exception et alors mettre la méthode divise dans un bloc try.

Programmation Orienté Objet
Cours 7

Mohamed Nabil Saidi

msaidi@insea.ac.ma

19 Avril 2012

Plan

- Les collections
- La généricité

Les collections

- Les collections sont des objets permettant de gérer des ensembles d'objets avec éventuellement la possibilité de gérer les doublons, les ordres de tri, etc.
- La version 1 de Java proposait:
  - java.util.Vector, java.util.Stack, java.util.Hashtable
  - Une interface java.util.Enumeration permettant de parcourir ces objets (elle offre deux méthodes : public abstract boolean hasMoreElements() et public abstract Object nextElement())

La classe Vector

- La classe Vector est un " tableau extensible " : on peut y stocker un nombre indéterminé d'objets (le nombre peut augmenter ou diminuer).
- On n'insère pas de variables de type primitifs (il faudrait les emballer dans des classes enveloppes : Integer, Double, ...)

Les méthodes de la classes Vector :

Vector()
Vector(int CapacitéInitiale)
Vector(int CapInit, int CapIncrement)
addElement(Object obj)
removeElement(Object obj)
firstElement()
lastElement()
isEmpty()

contains(Object obj)
size()
setSize()
indexOf (Object obj)
capacity()
elementAt(int index)

elements()

La classe Vector (2)

Remplissage d'un Vector

```
Vector vect=new Vector ();
for (int i=0; i < 10; i++ )
        vect.addElement("Element de type String numero" + i);
```

Parcours avec elementAt

```
for (int i=0; i < 10; i++ )
        System.out.println(vect.elementAt(i));
```

Parcours avec Enumeration

```
Enumeration enum=vect.elements();
while(enum.hasMoreElements())
        System.out.println(enum.nextElement());
```

Exemple de la collection "stack"

```
package teststack;
import java.util.*
public class ExempleStack
{
Stack pile ;
public ExempleStack ()
{
pile = new Stack () ;
pile.push("Je suis ") ; pile.push("Un exemple ") ; pile.push("de pile")
;
Enumeration enum = pile.elements () ;
while (enum.hasMoreElements()){
System.out.println (enum.nextElement()) ;
}
}
public static void main(String[ ] args){
new ExempleStack () ;
}
}
```

Je suis

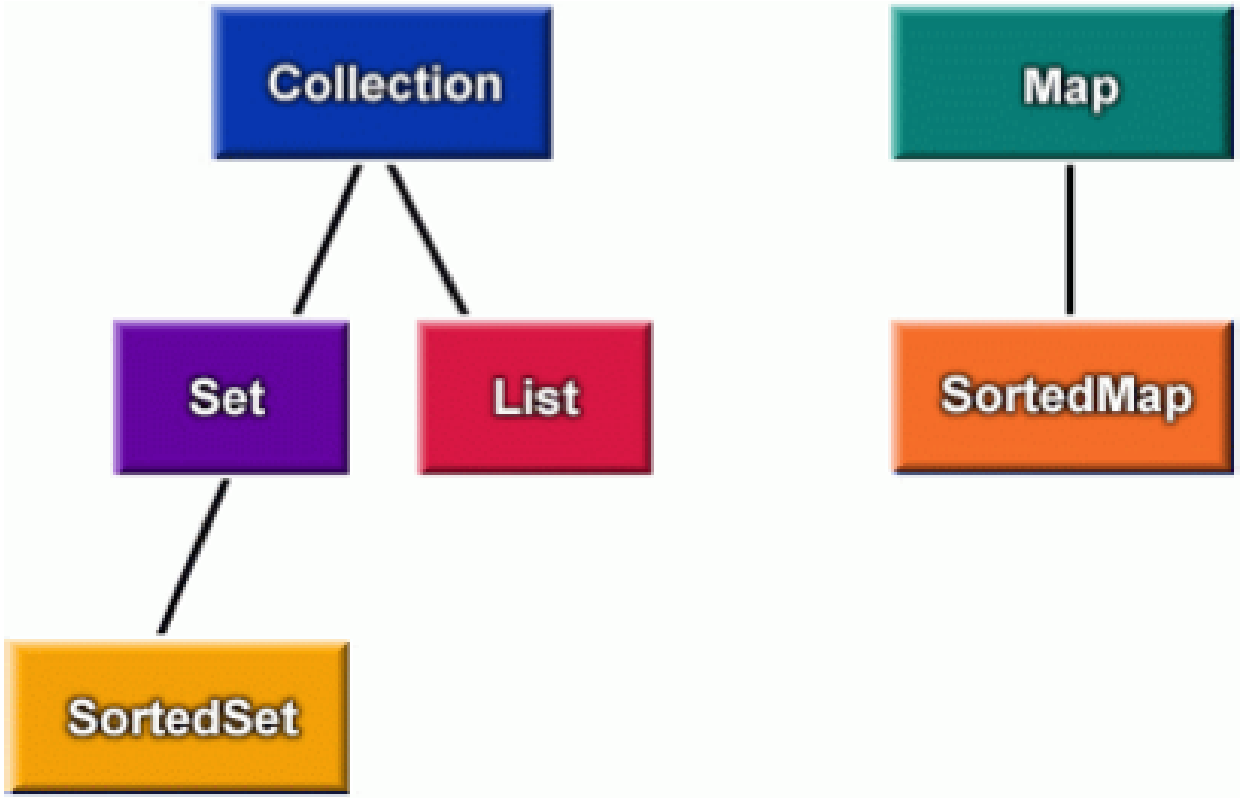
Un exemple

de pile

Interfaces de collections

Collection à partir de java 2

Réparties en deux groupes :



Collection à partir de java 2

- **Collection :** interface qui est implémentée par la plupart des objets qui gèrent des collections.
- **Map :** interface qui définit des méthodes pour des objets qui gèrent des collections sous la forme **clé/valeur**
- **Set :** interface pour des objets qui n'autorisent pas la gestion des doublons dans l'ensemble
- **List :** interface pour des objets qui autorisent la gestion des doublons et un accès direct à un élément
- **SortedSet :** interface qui étend l'interface Set et permet d'ordonner l'ensemble
- **SortedMap :** interface qui étend l'interface Map et permet d'ordonner l'ensemble

Implémentation des interfaces

Interface	Implémentation
Set	HashSet
SortedSet	TreeSet
List	ArrayList, LinkedList, Vector
Map	HashMap, Hashtable
SortedMap	TreeMap

La classe ArrayList

- Une instance de la classe ArrayList est une sorte de tableau qui peut contenir un nombre quelconque d'instances d'une classe quelconque.
- Les emplacements sont indexés par des nombres entiers (à partir de 0).
- Les constructeurs :
  - ArrayList()
  - ArrayList(int taille initiale) : peut être utile si on connaît la taille finale ou initiale

La classe ArrayList (2)

Les méthodes principales :

- boolean add(E elt)
- void add(int indice, E elt)
- boolean contains(Object obj)
- E get(int indice)
- int indexOf(Object obj)
- E remove(int indice)
- E set(int indice, E elt)
- int size()

La classe ArrayList (3)

TestArrayList.java

```
class TestArrayList {
public static void main (String [] args) {
ArrayList a = new ArrayList( );
a.add( "lundi" );
a.add( "mardi" );
a.add( "mercredi" );
a.add( "jeudi" );
for (int i=0; i < a.size(); i++ ) {
System.out.println( a.get(i) );
}
}
}
```

Interface Iterator

Testiterator.java

```
ArrayList a = new ArrayList( 100 );
a.add( "lundi" );
a.add( "mardi" );
a.add( "mercredi" );
a.add( "jeudi" );
for ( Iterator iter = a.iterator(); iter.hasNext(); )
{
String value = iter.next();
System.out.println( value );
}
```

La classe Hashtable

- La classe Hashtable implante l'interface Map (carte) : une carte enregistre des paires clé/valeur(objet)
- La recherche d'une valeur (objet) se fait donc à partir de la clé (et non selon un index comme dans le cas d'une table)
- Le dictionnaire est un exemple de carte : les clés correspondent aux mots et les objets associés aux significations
- Seuls les objets définissant les méthodes hashCode() et equals peuvent figurer comme clé dans une carte
- pour chacun des clés , Hashtable (table de hachage) utilise les fonctions de hachage pour calculer un nombre entier, appelé **code de hachage**.

La classe Hashtable (2)

Les méthodes principales :

- Les clés doivent être uniques : si on fait deux appels à put avec la même clé, la seconde valeur remplacera la première

Les méthodes principales :

- put(Object Cle, Object Valeur) //Ajouter un élément
- Object get(Object Cle) //Retrouver un objet (get retourne null si aucune paire ne se trouve avec une telle clé)
- remove(Object Cle) //supprime l'entrée correspondant à la clé

La classe Hashtable (3)

TestHashtable.java

```
class TestHashtable {
public static void main (String [] args) {
Hashtable dict = new Hashtable();
dict.put ("petit", new Rectangle(0,0,5,5));
dict.put ("moyen", new Rectangle(10,10,15,15));
dict.put ("grand", new Rectangle(20,20,25,25));
Rectangle r = (Rectangle) dict.get("moyen");
//Besoin de sous-casting, car get(..) retourne un object
}
}
```

Généricité

Collection et java 1.5

- Jusqu'à la version 1.4, on stockait et récupérait des "Object" d'une collection.
- **Exemple:**

```
ArrayList liste = new ArrayList ()
liste.add (new Maclasse ()) ;
Maclasse obj = (Maclasse) liste.get (0) ;
```

- Depuis la version 1.5, il est recommandé de spécifier la nature des objets stockés.
- **Exemple:**

```
ArrayList<Maclasse> liste = new
ArrayList<Maclasse> () ;
liste.add (new Maclasse ())
Maclasse obj = liste.get (0) ;
```

Les types génériques

- Les types génériques, définis par la JSR 14, permettent de spécifier le type d'objets que l'on va placer dans une collection d'objets (List, Vector)
- Avantages:
  - meilleure lisibilité: on connaît à la lecture du programme quel type d'objets seront placés dans la collection.
  - La vérification peut être fait à la compilation.
  - Le cast pour récupérer un objet de la collection est devenu implicite (sans cette fonctionnalité, il fallait faire un cast explicite, sachant que celui-ci peut échouer mais cela n'était détectable qu'à l'exécution)
- La syntaxe pour utiliser les types génériques utilise les symboles $<$ et $>$.

Exemple de type générique

```
import java.util.* ;
public class TestGenerique{
public static void main(String[ ] args) { new TestGenerique () ; }
public TestGenerique (){
String chaine,str ;
boolean bFinBoucle = false ;
List<String> liste = new ArrayList () ;
Scanner clavier = new Scanner (System.in) ;
while (bFinBoucle == false){
chaine = clavier.next () ;
if (chaine.equalsIgnoreCase("quit") == false)
liste.add (chaine) ; // on ne peut stocker que des Strings
else bFinBoucle = true ;
}
for (Iterator<String> iter = liste.iterator (); iter.hasNext () ;)
{
str = iter.next () ; // Pas de cast ici
System.out.println (str) ;
}
}
}
```

Les classes génériques

```
class MaclasseGenerique<T1,T2>{
private T1 param1 ;
private T2 param2 ;
public MaclasseGenerique (T1 param1,T2 param2)
{
this.param1 = param1 ;
this.param2 = param2 ;
}
public T1 getParam1 () { return param1 ; }
public T2 getParam2 () { return param2 ; }
}
public class TestclasseGenerique{
public TestclasseGenerique ()
{
new MaclasseGenerique<String,Integer> ("Dupont",33) ;
}
public static void main(String[ ] args)
{
new TestclasseGenerique () ;
}
}
```

Définition

- Une méthode peut être paramétrée avec des valeurs.
- La généricité permet de paramétrer du code avec des types de données.
- **Exemple :**

  class ArrayList<T>

- dont le code est paramétré par un type T
- Pour l'utiliser il faut passer un type en argument :
  new ArrayList<Employe>()

Pourquoi la généricité

- Une collection d'objets ne contient le plus souvent qu'un seul type d'objet : liste d'employés, liste de livres, liste de chaînes de caractères, etc.
- Mais sans la généricité les éléments des collections doivent être déclarés de type Object.
- Il est impossible d'indiquer qu'une collection ne contenait qu'un seul type d'objet, comme on le fait avec les tableaux (par exemple String[ ])

Exemple de code non générique

```
ArrayList employes = new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
.  .  .  // On ajoute d'autres employes
for(int i = 0; i < employes.size(); i++) {
System.out.println(((Employe)
employes.get(i)).getNom());
}
```

- Remarque que le casting est nécessaire pour pouvoir utiliser getNom().

Autre exemple (problématique)

```
ArrayList employes = new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
.  .  .  // On ajoute d'autres employes

// On ajoute un livre au milieu des employes
Livre livre = new Livre(...);
employes.add(livre);

for (int i = 0; i < employes.size(); i++) {
System.out.println(((Employe)
employes.get(i)).getNom());
}
```

Conséquences

- Il est impossible d'indiquer qu'une liste ne contient que des instances d'un certain type.
- Certaines erreurs ne peuvent être repérées qu'à l'exécution et pas à la compilation.
- Il faut sans arrêt caster les éléments pour pouvoir utiliser les méthodes qui n'étaient pas dans `Object`.
- Si un objet contenu dans les collections n'est pas du type attendu, on a une erreur à l'exécution mais pas à la compilation.

Collections à type particulier

- Si on veut éviter ces problèmes (éviter les casts et vérifier les types à la compilation), il faut
  - écrire un code différent pour chaque collection d'un type particulier
  - Par exemple, il faut écrire une classe ListInteger et une classe ListEmploye.
  - Ce qui revient à écrire plusieurs fois la même chose, en changeant seulement les types.
  - Ou alors,écrire des codes génériques.

Généricité

- La généricité permet de paramétrer une classe ou interface avec un ou plusieurs types de données.
- On peut par exemple donner en paramètre le type des éléments d'un ArrayList : ArrayList<**E**>
- **E** est un paramètre de type formel (ou plus simplement paramètre de type) ou variable de type.
- **E** sera remplacé par un argument de type pour typer des expressions ou créer des objets : ArrayList<Integer> l = new ArrayList<Integer>();

Généricité (2)

- ArrayList<E> est un type générique (plus exactement une classe générique).
- ArrayList<Employe> est une instanciation du type générique ArrayList<E>; c'est un type paramétré concrêt (plus exactement une classe paramétrée) dont l'argument de type est Employe.
- La généricité permet d'écrire des collections dont tous les éléments ont le même type, sans avoir à répéter plusieurs fois le même code en vérifiant le typage dès la compilation en évitant les casts à l'exécution.

Extraits de la classe ArrayList

```
public class ArrayList<E> extends abstractList<E>
{
public ArrayList() // pas ArrayList<E>() !
public boolean add(E element)
public E get(int index)
public E set(int index, E element)
}
```

Utilisation d'un ArrayList paramétré

```
ArrayList<Employe> employes = new
ArrayList<Employe>();

Employe e = new Employe("Dupond");
employes.add(e);
...  // On ajoute d'autres employes


for (int i = 0; i < employes.size(); i++)
{
System.out.println(employes.get(i).getNom());
}
```

Erreur détectée à la compilation

```
ArrayList<Employe> employes = new
ArrayList<Employe>();

Employe e = new Employe("Dupond");
employes.add(e);
...// On ajoute d'autres employes

// Ajoute un livre au milieu des employes
Livre livre = new Livre(...);
employes.add(livre); // Erreur de compilation
```

Quelques définitions

- Type générique : une classe ou une interface paramétrée par une section de paramètres de la forme <T1, T2, . . ., Tn>.
- Les Ti sont les paramètres de type formels. Ils représentent des types inconnus au moment de la compilation du type générique.
- On place la liste des paramètres à la suite du nom de la classe ou de l'interface :
  List<E>
  Map<K,V>.

Où peuvent apparaître les paramètres de type ?

- Dans le code du type générique, les paramètres de type formels peuvent être utilisés comme les autres types pour déclarer des variables, des paramètres, des tableaux ou des types retour de méthodes :
  - `E element;`
  - `E[ ] elements;`

Où peuvent apparaître les paramètres de type ? (2)

- Ils ne peuvent être utilisés pour créer des objets ou des tableaux, ni comme super-type d'un autre type :
  - pas de `new E()`
  - pas de `new E[10]`
  - pas de `class C extends E`
- On ne peut utiliser un paramètre de type à droite de `instanceof` : pas de x instanceof E

Types des arguments de type

- Les arguments de type peuvent être des classes, même abstraites, ou des interfaces ;
- **par exemple :** `new ArrayList<Comparable>`
- Ils peuvent même être des paramètres de type formels ;
- **par exemple :**

```
public class C<E>
{
...
f = new ArrayList<E>();
...
}
```

- Un argument de type ne peut pas être un type primitif.

Arguments de type abstraits

- On peut utiliser un argument de type abstrait (classe abstraite, interface) pour instancier une classe générique ;
- si `EmployeI` est une interface, **on peut écrire :**

```
List<EmployeI> l = new ArrayList<EmployeI>();
```

- évidemment, il faudra remplir cette liste avec des employés concrêts (les classes qui implémente l'interface EmployeI).

Création d'une instance de classe paramétrée

- Au moment de la création d'une classe paramétrée, on indique le type de la collection en donnant un argument de type pour chaque paramètre de type formel

```
ArrayList<String> liste = new
                          ArrayList<String>();

Map<String,Integer> map = new
                           Map<String,Integer>();
```

Utilisation des types génériques

- On peut utiliser les types génériques comme sur-type (classe mère ou interface).
- **Exemple :**

```
public class C<P> extends M<P>
```

- Sinon, on ne peut utiliser un paramètre de type que dans une classe générique paramétrée par ce paramètre de type.

Méthode générique

- Comme une classe ou une interface, une méthode (ou un constructeur) peut être paramétrée par un ou plusieurs types.

- Une méthode générique peut être incluse dans une classe non générique, ou dans une classe générique (si elle utilise un paramètre autre que les paramètres de type formels de la classe).

Syntaxe

- Une liste de paramètres apparaît dans l'en-tête de la méthode (juste avant le type de retour) pour indiquer que la méthode ou le constructeur dépend de types non connus au moment de l'écriture de la méthode : `<T1, T2, ... > ... m(...)`

- Exemple de l'interface Collection<E> :

```
public abstract <T> T[ ] toArray(T[ ] a)
```

Instanciation d'une méthode générique

- On peut appeler une méthode paramétrée en la préfixant par le (ou les) type qui doit remplacer le paramètre de type : `NomObjet.<String> NomMethode()`.

- Mais le plus souvent le compilateur peut faire une inférence de type ("deviner" le type) d'après le contexte d'appel de la méthode.

Inférence de type

- On appelle alors la méthode paramétrée sans la préfixer par une argument de type :

```
                    ArrayList<Personne> liste;
.   .   .
Employe[ ] res = liste.toArray(new Employe[0]);
// Inutile de préfixer par le type :
// liste.<Employe> toArray(...);
```

Inférence de type (2)

- Parfois, c'est un peu plus complexe pour le compilateur :

```
public <T> T choose(T a, T b) { ... }
...
Number n = choose(new Integer(0), new
Double(0.0));
```

- Le compilateur infère Number qui est la plus proche super-classe de `Integer` et `Double`.

Sous-typage et généricité

- Puisque que `ArrayList<E>` implémente `List<E>`, `ArrayList<E>` est un sous-type de `List<E>`.
- Ceci est vrai pour tous les types paramétrés construits à partir de ces 2 types.
- Par exemple, `ArrayList<Personne>` est un sous-type de `List<Personne>` ou `ArrayList<Integer>` est un sous-type de `List<Integer>`.

Sous-typage pour les types paramétrés

- **Mais attention :** si B hérite de A, les classes ArrayList<B> et ArrayList<A> n'ont aucun lien de sous-typage entre elles.

- **Exemple :** ArrayList<Integer> n'est pas un sous-type de ArrayList<Number>! ! (Number est la classe mère de Integer).

Raison

- Si ArrayList<B>était un sous-type de ArrayList<A>, le code suivant compilerait :

```
// ce code ne compile pas!!
                    ArrayList<A> la = new ArrayList<B>;
                    la.add(new A());
```

- Quel serait le problème ?
- Ce code autoriserait l'ajout dans un ArrayList<B> un élément qui n'est pas un B !

Conséquences

- Une méthode `sort(ArrayList<Number> aln)` qui trie un `ArrayList<Number>` ne peut être appliquée à un `ArrayList<Integer>`.

- De même, l'instruction suivante est interdite :

```
ArrayList<Number> l = new ArrayList<Integer>();
```

Exemple de problème

```
public static void printElements(ArrayList<Number>
liste) {
for(Number n :  liste) {
System.out.println(n.intValue());
}
}
ArrayList<Double> liste = new ArrayList<Double>();
...
printElements(liste); // Ne compile pas !
```

Types Joker

- Un type Joker sans limite: <?>
- Un type Joker limité aux sous-classes d'une classe <? extends Animal> ou aux classes qui implémentent une interface <? extends Comparable>
- Un type Joker limité aux classes ayant une certaine sous-classe: <? super Integer>
- Exemple d'un attribut : ArrayList <? extends Vehicule> liste;

Méthodes ayant un nombre variable de paramètres de même type

Exemple.java

```
public class Exemple.java {
static void ecrireLesMots(String ... mots) {
for (String mot : mots) // mots est envisagé comme un tableau
System.out.print(mot + " ");
}
public static void main (String [] args) {
ecrireMots(" bonjour ", " le ", " monde ");
}
}
```