


# Mise à niveau Java

Yohan Boichut & Frédéric Moal



# Plan

1. Introduction
2. syntaxe, types primitifs
3. classes et objets
4. tableaux
5. String
6. exceptions
7. Héritage, polymorphisme, interfaces

# Conventions

# commande : une commande à taper dans un terminal

slides sur fond gris : exercices à faire sur machine

# Introduction

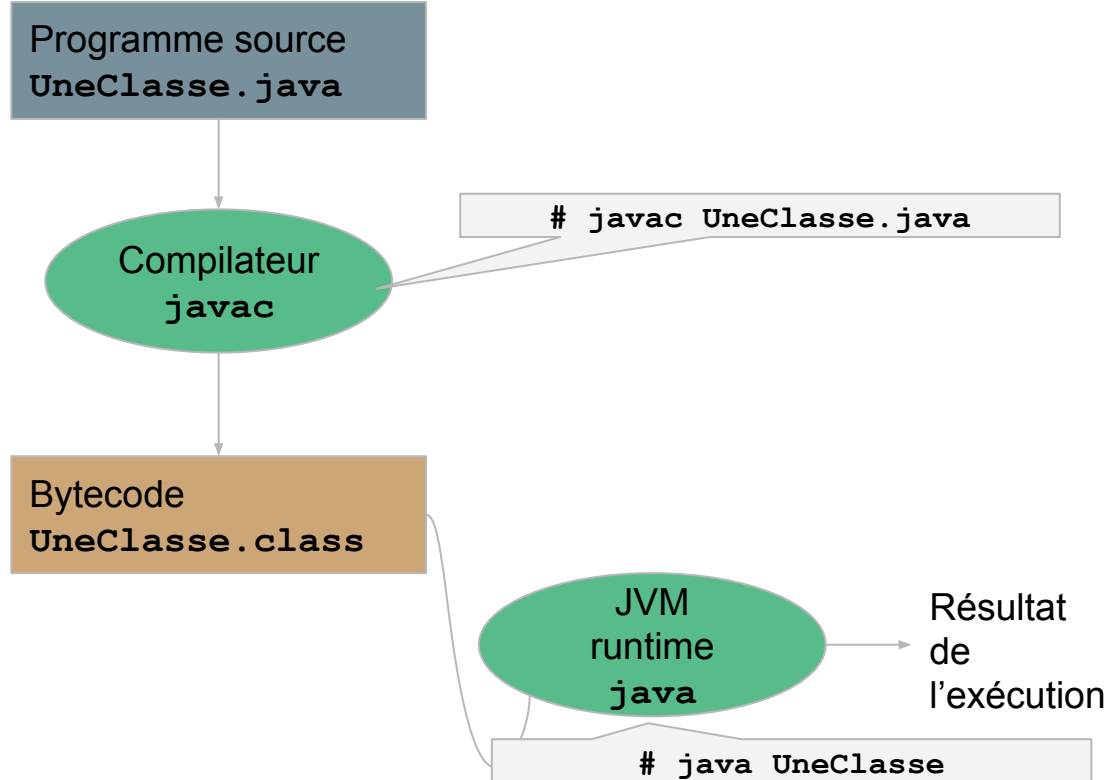
# Le langage Java

- Langage orienté objet, à classes (les objets sont décrits/regroupés dans des classes)
- de syntaxe proche du langage C
- fourni avec le JDK (Java Development Kit) :
  - outils de développement
  - ensemble de paquetages très riches et très variés
- portable grâce à l'exécution par une machine virtuelle : « Write once, run everywhere »

# Compilation en bytecode puis exécution

Programme écrit en Java

Programme en bytecode,  
indépendant de l'ordinateur



# Le premier programme

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

Rem : La classe Helloworld est public, donc le fichier qui la contient doit s'appeler (en tenant compte des majuscules et minuscules) HelloWorld.java

# Compilation avec javac

- Oracle fournit le compilateur javac avec le JDK

```
javac HelloWorld.java
```

crée un fichier « HelloWorld.class » qui contient le bytecode [sans option, il situé dans le même répertoire que le fichier « .java »]

- Le fichier à compiler peut être désigné par un chemin absolu ou relatif :

```
javac util/Liste.java
```



# Exécution du bytecode avec le JDK (JRE)

- Oracle fournit le programme java qui simule une JVM permettant d'exécuter du bytecode

**java HelloWorld**



*Nom d'une **classe** :  
pas de suffixe .class !*

exécute le bytecode de la méthode main de la classe HelloWorld

- HelloWorld est un nom de classe et pas un nom de fichier :
  - on ne peut pas donner un chemin
  - pas de suffixe .class

# Spécifications

- Java, c'est en fait
  - le langage Java : <http://java.sun.com/docs/books/jls/>
  - une JVM : <http://java.sun.com/docs/books/vmspec/>
  - les API : selon la documentation javadoc fournie avec les différents paquetages
- Java n'est pas normalisé ; son évolution est gérée par le JCP (Java Community Process ; <http://www.jcp.org/>) dans lequel Oracle tient une place prépondérante

# 3 éditions, plusieurs versions

- Java SE : Java Standard Edition ; JDK = Java SE Development Kit
  - Java EE : Enterprise Edition qui ajoute les API pour écrire des applications installées sur les serveurs dans des applications distribuées : servlet, JSP, JSF, EJB,...
  - Java ME : Micro Edition, version pour écrire des programmes embarqués (carte à puce/Java card, téléphone portable,...)
- 
- Versions : actuellement Java SE 8 ; Java est passé directement de la version 1.4 à la version 5.0, puis 6, 7, 8...

# Notre environnement de travail

- Éditeur de texte (...)
- Compilateur (javac)
- Interpréteur de bytecode (java)
- Aide en ligne sur le JDK (sous navigateur Web)
- ...
- un IDE : IntelliJ <https://www.jetbrains.com/idea/>
  - version gratuite [Community] et payante [Ultimate], licences accessibles dans les salles de TP de la fac
  - pour chez vous : <https://www.jetbrains.com/shop/eform/students> inscription avec votre mail de l'Université

# Bibliographie

- Tutos d'Oracle : <https://docs.oracle.com/javase/tutorial/>
- Thinking in Java (<http://mindview.net/Books/TIJ/>) / Penser en Java (<http://penserenjava.free.fr/>)
- Livre : Core Java, Prentice-Hall (vol. 1) / Au coeur de Java, Campus Press
- Livre : Java in a Nutshell, O'Reilly
- Livre : Java tête la première / Head First Java, O'Reilly
- Le doudoux (<http://www.jmdoudoux.fr/>, 3413 pages)
- Internet :
  - developpez.com
  - dzone.com [refcardz]
  - slides de Richard Grin, Miage Nice
  - les cast codeurs
  - java code geek, java ranch, javaworld, InfoQ, ... Stack Overflow !
  - Javarevisited (<http://javarevisited.blogspot.fr>)
  - twitter / youtube (devox FR)

# Structures lexicales

# Identificateurs

- Un identificateur (nom de variable, constante, ... ) en Java
  - est de longueur quelconque
  - commence par une lettre Unicode (caractères ASCII recommandés)
  - peut ensuite contenir des lettres ou des chiffres ou le caractère souligné « \_ »
  - ne doit pas être un mot-clé ou un des constantes : true, false ou null
- Par convention :
  - nom de variables en minuscules : total
  - les mots séparés par des majuscules : monTotalFinal
  - nom d'une classe commence par une Majuscule : MaPremiereClasse
  - évitez les accents !
  - nom des constantes (final) en majuscules : VALEUR
  - séparation par \_ : VALEUR\_PAR\_DEFAULT

# Mots clés Java

abstract, boolean, break, byte, case, catch, char, class, const\*, continue, default, do, double, enum\*\*, else, extends, final, finally, float, for, goto\*, if, implements, import, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, static, strictfp, super, switch, synchronized, this, throw, throws, transient, try, void, volatile, while

\*: pas encore utilisé

\*\* : depuis Java SE 5



# Commentaires

- Sur une seule ligne :

```
// Voici un commentaire  
int prime = 1500; // prime fin de mois
```

- Sur plusieurs lignes :

```
/* Première ligne du commentaire  
suite du commentaire */
```

- Documentation automatique par javadoc (classes et méthodes)

```
/**  
 * Teste l'égalité de deux Complexes.  
 * @param c1      Le premier complexe  
 * @param c2      Le deuxième complexe  
 * @return vrai (true) si les complexes sont égaux ou faux (false) sinon  
 */
```



# Types de données



# 2 grands types de données

Toutes les données manipulées par Java ne sont pas des objets

- 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 [pointeurs] aux objets

# Types primitifs

- **boolean** (true/false)
- Nombres entiers : **byte** (1 octet, -128 à 127), **short** (2 octets, -32.768 à 32.767), **int** (4 octets, -2.147.483.648 et 2.147.483.647), **long** (8 octets, +ou-  $9,2 \times 10^{18}$ )
- Nombres non entiers, à virgule flottante : **float** (4 octets, de  $1,4 \times 10^{-45}$  à  $3,4 \times 10^{38}$ , 7 chiffres), **double** (8 octets, de  $4,9 \times 10^{-324}$  à  $1,8 \times 10^{308}$ , 17 chiffres).
- Caractère (un seul) : **char** (2 octets) ; codé par le codage Unicode (pas ASCII)

# Valeurs numériques

- Une constante « entière » est de type long si elle est suffixée par « L » et de type int sinon ; on peut insérer des \_ entre les blocs de chiffres
- Une constante « flottante » est de type float si elle est suffixée par « F » et de type double sinon

35

2589L // constante de type long

1\_567\_688

456.7 ou 4.567e2 // 456,7 de type double

.123587E-25F // de type float

012 // 12 en octal = 10 en décimal

0xA7 // A7 en hexa = 167 en décimal

0x50\_45\_14\_56

0b01010000010001010001010001010110 // en binaire

0b01010000\_01000101\_00010100\_01010110

# Autres valeurs

- Valeurs de type char

- Un caractère Unicode entouré par « ' »
- CR et LF interdits (caractères de fin de ligne)

`'A'`

`'\t' '\n' '\r' '\\' '\'' '\"'`

`'\u20ac'` (`\u` suivi du code Unicode hexadécimal d'un caractère ;  
représente €)

`'α'`

- Type booléen : **false** **true**
- Référence inexistante ; indique qu'une variable de type objet ne référence rien : **null**

# Valeurs par défaut

Si elles ne sont *pas initialisées*, les variables d'instance ou de classe (pas les variables locales d'une méthode) reçoivent par défaut les valeurs suivantes :

boolean	false
char	'\u0000'
Entier (byte short int long)	0 0L
Flottant (float double)	0.0F 0.0D
Référence d'objet	null

# Opérateurs sur les types primitifs

- Les plus utilisés :

= + - \* / % ++ -- += -= \*= /=

== != > < >= <=

&& || ! (et, ou, négation)

- x++ : la valeur actuelle de x est utilisée dans l'expression et juste après x est incrémenté
- ++x : la valeur de x est incrémentée et ensuite la valeur de x est utilisée

x = 1; y = ++x \* ++x; // Que vaut y ?



# Exemples

```
int x = 9, y = 2;
```

```
int z = x/y;      // z = 4 (division entière)
```

```
z = x++ / y;      // z = 4 puis x = 10
```

```
z = --x / y;      // x = 9 puis z = 4
```

```
if (z == y)       // et pas un seul "=" !
```

```
    x += y;       // x = x + y
```

```
if (y != 0 && x/y > 8)    // raccourci, pas d'erreur si y = 0
```

```
    x = y = 3;          // y = 3, puis x = y, (à éviter !)
```

# Résultat d'un calcul

- Tout calcul entre entiers donne un résultat de type int (ou long si au moins un des opérandes est de type long)
- Dépassement possibles !

`System.out.print(2000000000 + 2000000000);` affiche -294967296

- Les types numériques « flottants » (non entiers) respectent la norme IEEE 754
- Erreurs de calculs possibles :

16.8 + 20.1 donne 36.9000000000000006

- Pour les traitements de comptabilité, utiliser `java.math.BigDecimal`

# Opérateurs et Priorités

Postfixés	[] . (params) expr++ expr--
Unaires	++expr --expr +expr -expr ~ !
Création et cast	new (type)expr
Multiplicatifs	* / %
Additifs	+ -
Décalages bits	<< >> >>>
Relationnels	< > <= >= instanceof
Egalité	== !=
Bits, et	&
Bits, ou exclusif	^
Bits, ou inclusif	
Logique, et	&&
Logique, ou	
Conditionnel	?:
Affectation	= += -= *= /= %= &= ^=  = <<= >>= >>>=

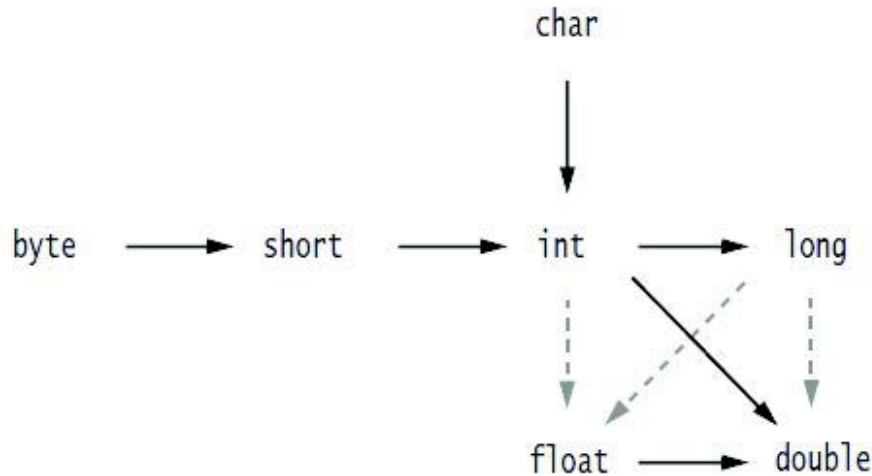
Les opérateurs d'égales priorités sont évalués de gauche à droite, sauf les opérateurs d'affectation, évalués de droite à gauche

Attention à la division /  
entier / entier : division entière  
sinon division double (choix par cast)

```
x = 1; y = ++x * ++x;  
Que vaut y ?
```

# Cast (transtypage) : changer un type primitif

- Un cast entre types primitifs peut occasionner une perte de données (eg int vers short)
- Un cast peut provoquer une simple perte de précision



# Cast implicite

- Une affectation entre types primitifs peut utiliser un cast implicite si elle ne peut pas provoquer de perte de valeur
- Par exemple, un short peut être affecté à un int sans nécessiter de cast explicite :

```
short s = ...;  
int i = s;
```

- ce cast implicite peut être effectué par le compilateur pour les paramètres d'un appel de méthode ; cf Jeu suivant

# Cast explicite

- Pour forcer la division / :

```
int x = 10, y = 3;
```

```
int z = x / y           // z=3
```

```
double zd = (double)x / y; // dz=3.333.. cast de x suffit
```

- Si une affectation peut provoquer une perte de valeur, elle doivent comporter un cast explicite :

```
int i = ...;
```

```
short s = (short)i;
```

- L'oubli de ce cast explicite provoque une erreur à la compilation  
Il faut écrire « `float f = 1.2f;` » et pas « `float f = 1.2;` »

# Problèmes

perte de précision sans cast : risque d'erreur

```
long l1 = 123456789;
long l2 = 123456788;
float f1 = l1;
float f2 = l2;
System.out.println(f1); // 1.23456792E8
System.out.println(f2); // 1.23456784E8
System.out.println(l1 - l2); // 1
System.out.println(f1 - f2); // 8 !
```

cast explicite : risque d'erreur sans aucun avertissement ni message

```
int i = 130;
b = (byte)i; // b = -126 !
int c = (int)1e+30; // c = 2147483647 !
```

# Un jeu !

```
int calcArea(int height, int width) {  
    return height * width;  
}
```

Lesquelles de ces expressions  
compilent et s'exécutent ?

1. `int a = calcArea(7, 12);`
2. `short c = 7;`
3. `calcArea(c,15);`
4. `int d = calcArea(57);`
5. `calcArea(2,3);`
6. `long t = 42;`
7. `int f = calcArea(t,17);`
8. `int g = calcArea();`
9. `calcArea();`
10. `byte h = calcArea(4,20);`
11. `int j = calcArea(2,3,5);`





# Les instructions



# Blocs

- Les méthodes sont structurées en blocs par les structures de la programmation structurée
  - suites de blocs
  - alternatives
  - répétitions
- Un bloc est un ensemble d'instructions délimité par { et }
- Les blocs peuvent être emboîtés les uns dans les autres

# Blocs & Portée des identificateurs

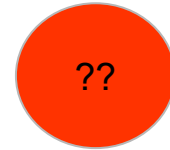
- Les blocs définissent la portée des identificateurs
- La portée d'un identificateur commence à l'endroit où il est déclaré et va jusqu'à la fin du bloc dans lequel il est défini, y compris dans les blocs emboîtés
- Les variables locales peuvent être déclarées n'importe où dans un bloc
- On peut aussi déclarer la variable qui contrôle une boucle « for » dans l'instruction « for » (la portée est la boucle) :

```
for (int i = 0; i < 8; i++) {  
    s += valeur[i];  
}
```

# Blocs & Portée des identificateurs

- Attention ! Java n'autorise pas la déclaration d'une variable dans un bloc avec le même nom qu'une variable d'un bloc emboîtant, ou qu'un paramètre de la méthode

```
int somme(int init) {  
    int i = init;  
    int j = 0;  
    for (int i=0; i<10; i++) {  
        j += i;  
    }  
    int init = 3;  
}
```



# Conditionnelle

if (expressionBooléenne)  
    bloc-instructions ou instruction  
[else  
    bloc-instructions ou instruction]

```
int x = y + 5;  
if (x % 2 == 0) {  
    type = 0;  
    x++;  
}  
else type = 1;
```

## Expression conditionnelle

test ? expression1 : expression2

```
int y = (x%2==0) ? x+1 : x;
```

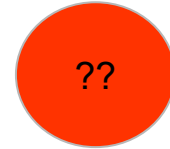
est équivalent à

```
int y;  
if (x % 2 == 0)  
    y = x + 1  
else  
    y = x;
```

# Conditionnelle : bonne pratique

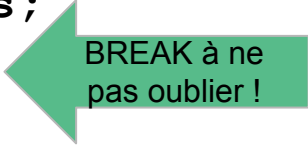
Lorsque plusieurs if sont emboîtés les uns dans les autres, un bloc else se rattache au dernier bloc if qui n'a pas de else !

```
x = 3;  
y = 8;  
if (x == y)  
if (x > 10)  
x = x + 1;  
else  
x = x + 2;
```



# Switch

```
switch(expression) {  
    case val1: instructions;  
                break;  
    ...  
    case valn: instructions;  
                break;  
    default:   instructions;  
}
```



BREAK à ne  
pas oublier !

expression est de type char, byte, short, ou int, ou énumération ou String

- S'il n'y a pas de clause default, rien n'est exécuté si expression ne correspond à aucun case

# Exemple switch

```
char lettre;  
int nbVoyelles = 0, nbA = 0,  
nbT = 0, nbAutre = 0;  
. . .  
switch (lettre) {  
    case 'a' : nbA++;  
    case 'e' : // pas d'instruction !  
    case 'i' : nbVoyelles++;  
                break;  
    case 't' : nbT++;  
                break;  
    default :  nbAutre++;  
}
```



# Répétition - boucles while

2 formes :

```
while (expressionBooléenne)  
    bloc-instructions ou instruction
```

```
do  
    bloc-instructions ou instruction  
while (expressionBooléenne)
```

# Exemple while

```
public class Diviseur {  
    public static void main(String[] args) {  
        int i = Integer.parseInt(args[0]);  
        int j = 2;  
        while (i % j != 0) {  
            j++;  
        }  
        System.out.println("PPD de "  
            + i + " : " + j);  
    }  
}
```

# Répétition - for

```
for(init; test; incrément){  
    instructions;  
}
```

est équivalent à

```
init;  
while (test) {  
    instructions;  
    incrément;  
}
```

```
int somme = 0;  
for (int i = 0; i <  
    tab.length; i++) {  
    somme += tab[i];  
}  
System.out.println(somme);
```

# Répétition - for each

- Une syntaxe pour simplifier le parcours d'un tableau/collection
- Attention, on ne dispose pas de la position dans le tableau (pas de « variable de boucle »)

```
String[] noms = new String[50];  
...  
// Lire « pour chaque nom dans noms »  
// « : » se lit « dans »  
for (String nom : noms) {  
    System.out.println(nom);  
}
```

# Répétition - break/continue

- break sort de la boucle et continue après la boucle
- continue passe à l'itération suivante
  
- break et continue peuvent être suivis d'un nom d'étiquette qui désigne une boucle englobant la boucle où elles se trouvent (une étiquette ne peut se trouver que devant une boucle) [bof !]

# Répétitions - exemple

```
int somme = 0;
for (int i = 0; i < tab.length; i++) {
    if (tab[i] == 0) break;
    if (tab[i] < 0) continue;
    somme += tab[i];
}
System.out.println(somme);
```

Qu'affiche ce code avec le tableau :  
1 ; -2 ; 5 ; -1 ; 0 ; 8 ; -3 ; 10 ?

# Un petit puzzle

# java Melange

a-b c-d

```
if (x == 1) {  
    System.out.print("d");  
    x = x - 1;  
}
```

```
if (x == 2) {  
    System.out.print("b c");  
}
```

```
class Melange {  
    public static void main(String [] args) {
```

```
        if (x > 2) {  
            System.out.print("a");  
        }
```

```
        int x = 3;
```

```
        while (x > 0) {
```

```
            x = x - 1;  
            System.out.print("-");
```

# Jeu 2 : joue au compilo et debug

```
class Exercice1 {  
    public static void main(String [] args) {  
        int x = 1;  
        while ( x < 10 ) {  
            if ( x > 3) {  
                System.out.println("big x");  
            }  
        }  
    }  
}
```



## Jeu 2 : même chose sur ce code

```
public static void main(String [] args) {  
    int x = 5;  
    while ( x > 1 ) {  
        x = x - 1;  
        if ( x < 3) {  
            System.out.println("small x");  
        }  
    }  
}
```

# Jeu 2 : Et encore...

```
class Exercice3{  
    int x = 5;  
    while ( x > 1 ) {  
        x = x - 1;  
        if ( x < 3) {  
            System.out.println("small x");  
        }  
    }  
}
```

# Jeu 3 : le bloc manquant ! cf slide suivant

```
class Test {  
    public static void main(String [] args) {  
        int x = 0;  
        int y = 0;  
        while ( x < 5 ) {  
            Bloc manquant  
            System.out.print(x + " " + y + " ");  
            x = x + 1;  
        }  
    }  
}
```

# Jeu 3 : bloc manquant : lequel fait quoi ?

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;  
if( y > 4 ) {  
y = y - 1;  
}
```

```
x = x + 1;  
y = y + x;
```

```
if ( y < 5 ) {  
    x = x + 1;  
    if ( y < 3 ) {  
        x = x - 1;  
    }  
}  
y = y + 2;
```

avec chaque bloc,  
ça donne ??

22 46

11 34 59

02 14 26 38

02 14 36 48

00 11 21 32 42

11 21 32 42 53

00 11 23 36 410

02 14 25 36 47



# Classes et Objets en Java

Mise à niveau Java  
Y. Boichut & F. Moal



# Paradigme Objet

- La programmation objet est un paradigme, une manière de « modéliser le monde » :
  - des objets ayant un état interne et un comportement
  - collaborent en s'échangeant des messages (pour fournir les fonctionnalités que l'on demande à l'application)
  - Java, SmallTalk, C++, C#, ...
- D'autres paradigmes :
  - programmation impérative (Pascal, C)
  - programmation fonctionnelle (Scheme, Lisp, "Java" )
  - programmation logique (prolog)
  - ...

# Orienté Objets

- Manipule des objets
- les programmes sont découpés suivant les types des objets manipulés
- les données sont regroupées avec les traitements qui les utilisent

Toute entité identifiable, concrète ou abstraite, peut être considérée comme un objet

- Un objet réagit à certains messages qu'on lui envoie de l'extérieur ; la façon dont il réagit détermine le comportement de l'objet
- Il ne réagit pas toujours de la même façon à un même message ; sa réaction dépend de l'état dans lequel il est

# Un objet en Java

- Un objet possède
  - une adresse en mémoire (identifie l'objet)
  - un comportement (ou interface)
  - un état interne
- L'état interne est donné par des valeurs de variables (variables d'instances)
- Le comportement est défini par des fonctions ou procédures, appelées méthodes



# Interactions entre objets

- Les objets interagissent en s'envoyant des messages synchrones
- Les méthodes de la classe d'un objet correspondent aux messages qu'on peut lui envoyer : quand un objet reçoit un message, il exécute la méthode correspondante

```
objet1.decrisToi() ;
```

```
employe.setSalaire(20000) ;
```

```
voiture.demarre() ;
```

```
voiture.vaAVitesse(50) ;
```

# Classe : regrouper les objets

- Les objets qui collaborent dans une application sont souvent très nombreux
- Mais on peut le plus souvent dégager des types d'objets : des objets ont une structure et un comportement très proches, sinon identiques (eg tous les livres dans une application de gestion d'une bibliothèque)
- La notion de classe correspond à cette notion de types d'objets

# Rôles d'une classe

- Une classe est
  - un type qui décrit une structure (variables d'instances) et un comportement (méthodes)
  - un module pour décomposer une application en entités plus petites
  - un générateur d'objets (par ses constructeurs)
- Une classe permet d'encapsuler les objets : les membres public sont vus de l'extérieur mais les membres private sont cachés

# Éléments d'une classe

- Les constructeurs (il peut y en avoir plusieurs) servent à créer les instances (les objets) de la classe
- Quand une instance est créée, son état est conservé dans les variables d'instance
- Les méthodes déterminent le comportement des instances de la classe quand elles reçoivent un message
- Les variables et les méthodes s'appellent les membres de la classe

# Classe : exemple

```
public class Livre {  
    private String titre, auteur;           // variables d'instance  
    private int nbPages;  
  
    public Livre(String unTitre, String unAuteur) {    // Constructeur  
        titre = unTitre;  
        auteur = unAuteur;  
    }  
    public String getAuteur() {                // méthode: accesseur  
        return auteur;  
    }  
    public void setNbPages(int nb) {           // méthode: modificateur  
        nbPages = nb;  
    }  
}
```

# Conventions de nommage

- Les noms de classes commencent par une majuscule (ce sont les seuls avec les constantes) : Cercle, Object
- Les mots contenus dans un identificateur commencent par une majuscule : UneClasse, uneMethode, uneAutreVariable
- Les constantes sont en majuscules avec les mots séparés par le caractère souligné « \_ » : UNE\_CONSTANTE
- Si possible, des noms pour les classes et des verbes pour les méthodes : Chien, aboyer()

# class en Java

- Une unique classe public par fichier
- Cette classe porte le même nom (commençant par une majuscule) que le fichier .java dans lequel elle est sauvée (public class Livre dans Livre.java)
- Il est possible d'ajouter d'autres classes (sans le mot clé public) dans le même fichier, mais elles ne seront accessibles que dans le package où elles sont définies

# Opérateur isinstance

- La syntaxe est :

objet isinstance nomClasse

- Exemple : **if (x isinstance Livre) { ...**
- Le résultat est un booléen :
  - true si x est de la classe Livre
  - false sinon
- cf compléments sur isinstance dans le cours sur l'héritage



# Constructeurs

# Classes / instances

- Une instance d'une classe est créée par un des constructeurs de la classe
- Une fois qu'elle est créée, l'instance
  - a son propre état interne (les valeurs des variables d'instance)
  - partage le code qui détermine son comportement (les méthodes) avec les autres instances de la classe
- Chaque classe a un ou plusieurs constructeurs qui servent à
  - créer les instances
  - initialiser l'état de ces instances
- Un constructeur
  - a le même nom que la classe
  - n'a pas de type retour

# Exemple de création d'instance

```
public class Employe {
    private String nom, prenom;
    private double salaire;
    // Constructeur
    public Employe(String n, String p) {
        nom = n;
        prenom = p;
    }
    . . .
    public static void main(String[] args) {
        Employe e1;
        e1 = new Employe("Dupond", "Pierre");
        e1.setSalaire(1200);
        . . .
    }
}
```

# Plusieurs constructeurs

```
public class Employe {
    private String nom, prenom;
    private double salaire;
    // 2 Constructeurs : surcharge
    public Employe(String n, String p) {
        nom = n;
        prenom = p;
    }
    public Employe(String n, String p, double s) {
        nom = n;
        prenom = p;
        salaire = s;
    }
    ...
    e1 = new Employe("Dupond", "Pierre");
    e2 = new Employe("Durand", "Jacques", 1500);
}
```

# Plusieurs constructeurs : this

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
    // 2 Constructeurs : appel du second par this  
    public Employe(String n, String p) {  
        this(n, p, 0.0);  
    }  
    public Employe(String n, String p, double s) {  
        nom = n;  
        prenom = p;  
        salaire = s;  
    }  
    ...  
    e1 = new Employe("Dupond", "Pierre");  
    e2 = new Employe("Durand", "Jacques", 1500);  
}
```

# Plusieurs constructeurs : this

```
public class Employe {
    private String nom, prenom;
    private double salaire;
    // 2 Constructeurs : appel du second par this
    private static final double DEFAULT_SALAIRE = 0.0;
    public Employe(String n, String p) {
        this(n, p, DEFAULT_SALAIRE);
    }
    public Employe(String n, String p, double s) {
        nom = n;
        prenom = p;
        salaire = s;
    }
    ...
    e1 = new Employe("Dupond", "Pierre");
    e2 = new Employe("Durand", "Jacques", 1500);
}
```

# Constructeur par défaut

- Lorsque le code d'une classe ne comporte pas de constructeur, un constructeur sera automatiquement ajouté par Java
- Pour une classe Classe, ce constructeur par défaut sera :

```
[public] Classe() { }
```

- Il est conseillé de toujours écrire au moins un constructeur !

# Méthodes



# Méthodes / Accesseurs

Deux types particuliers de méthodes servent juste à donner accès aux variables depuis l'extérieur de la classe :

- les accesseurs en lecture pour lire les valeurs des variables ; « accesseur en lecture » est souvent abrégé en « accesseur » ; getter en anglais
- les accesseurs en écriture, ou modificateurs, ou mutateurs, pour modifier leur valeur ; setter en anglais

Les autres types de méthodes permettent aux instances de la classe d'offrir des services plus complexes aux autres instances

Enfin, des méthodes (private) servent de « sous-programmes » utilitaires aux autres méthodes de la classe

# Paramètres des méthodes

- Souvent les méthodes ou les constructeurs ont besoin qu'on leur passe des données initiales sous la forme de paramètres
- On doit indiquer le type des paramètres dans la déclaration de la méthode :

```
setSalaire(double unSalaire)
```

```
calculerSalaire(int indice, double prime)
```

- Quand la méthode ou le constructeur n'a pas de paramètre, on ne met rien entre les parenthèses :

```
getSalaire()
```

# Type de retour d'une méthode

- Quand la méthode renvoie une valeur, on doit indiquer le type de la valeur renvoyée dans la déclaration de la méthode :

```
double calculSalaire(int indice, double prime)
```

- Le pseudo-type void indique qu'aucune valeur n'est renvoyée :

```
void setSalaire(double unSalaire)
```

# Exemple

```
public class Employe {  
    . . .  
    public void setSalaire(double unSalaire) {  
        if (unSalaire >= 0.0)  
            salaire = unSalaire;  
    }  
    public double getSalaire() {  
        return salaire;  
    }  
    public boolean accomplir(Tache t) {  
        . . .  
    }  
}
```

# Surcharge d'une méthode

- En Java, on peut surcharger une méthode, c'est-à-dire, ajouter une méthode qui a le même nom mais pas la même signature qu'une autre méthode :

```
calculerSalaire(int)
```

```
calculerSalaire(int, double)
```

- Il est interdit de surcharger une méthode en changeant seulement le type de retour

```
int calculerSalaire(int)
```

```
double calculerSalaire(int)
```

# Variables

# Natures des variables

- Les variables d'instances
  - sont déclarées en dehors de toute méthode
  - conservent l'état d'un objet, instance de la classe
  - sont accessibles et partagées par toutes les méthodes de la classe
- Les variables locales
  - sont déclarées à l'intérieur d'une méthode
  - conservent une valeur utilisée pendant l'exécution de la méthode
  - ne sont accessibles que dans le bloc dans lequel elles ont été déclarées
- Laquelle choisir pour une variable de type objet ?

Si cet objet est utilisé par plusieurs méthodes de la classe, l'objet devra être référencé par une variable d'instance

# Déclaration des variables

- Toute variable doit être déclarée avant d'être utilisée
- Déclaration d'une variable : on indique au compilateur que le programme va utiliser une variable de ce nom et de ce type

```
double prime;
```

```
Employe e1;
```

```
Point centre;
```



# Affectation

- L'affectation d'une valeur à une variable est effectuée par l'instruction

`variable = expression;`

- L'expression est calculée et ensuite la valeur calculée est affectée à la variable

Exemple :

`x = 3;`

`x = z + 1;`

`livre = null;`

# Initialisation

- Une variable doit être initialisée avant d'être utilisée dans une expression
- Si elles ne sont pas initialisées par le programmeur, les variables d'instance (et les variables de classe étudiées plus loin) reçoivent les valeurs par défaut de leur type (0 pour les types numériques, null pour les objets)
- L'utilisation d'une variable locale non initialisée par le programmeur provoque une erreur (pas d'initialisation par défaut) à la compilation
- On peut initialiser une variable en la déclarant

```
double prime = 200.0;
```

```
Employe e1 = new Employe("Dupond", "Jean");
```

```
double salaire = prime + 500.0;
```

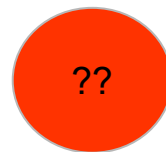
# Déclaration / Initialisation

- Il ne faut pas confondre
  - déclaration d'une variable
  - création d'un objet référencé par cette variable

« Employe e1; »

déclare que l'on va utiliser une variable e1 qui référencera un objet de la classe Employe, mais aucun objet n'est créé

```
Employe e1;  
e1.setSalaire(1200);  
...
```



# Désignation des variables d'instance

- Soit un objet o1 ; la valeur d'une variable v de o1 est désignée par o1.v

```
Cercle c1 = new Cercle(p1, 10);
```

```
System.out.println(c1.rayon); // affiche 10
```

- Remarque : le plus souvent les variables sont private et on ne peut pas y accéder directement en dehors de leur classe

# Variables d'instance final

Une variable d'instance final est constante pour chaque instance ; mais elle peut avoir 2 valeurs différentes pour 2 instances

Une variable d'instance final peut ne pas être initialisée à sa déclaration mais elle doit avoir une valeur à la sortie de tous les constructeurs



Accès aux membres



# Plusieurs autorisation d'accès aux membres/var

- **private** : seule la classe dans laquelle il est déclaré a accès (à ce membre ou constructeur)
- **public** : toutes les classes sans exception y ont accès
- Sinon, par défaut, seules les classes du même paquetage que la classe dans lequel il est déclaré y ont accès (un paquetage est un regroupement de classes ; cf plus loin dans le cours)
- **protected** cf cours sur l'héritage

# Droits associés à la classe pas à l'objet

En Java, la protection des attributs se fait classe par classe, et pas objet par objet

Un objet a accès à tous les attributs d'un objet de la même classe, même les attributs privés !

```
public class Entier {  
    private int laValeur;  
    private void plus(Entier autreEntier) {  
        laValeur = laValeur + autreEntier.laValeur;  
    }  
}
```



# Bonnes pratiques pour les droits

[Autant que possible] l'état interne d'un objet (les variables d'instance) doit être private

Si on veut autoriser la lecture d'une variable depuis l'extérieur de la classe, on lui associe un accesseur, avec le niveau d'accessibilité que l'on veut

Si on veut autoriser la modification d'une variable, on lui associe un modificateur, qui permet la modification tout en contrôlant la validité de la modification

this

# this

- Le code d'une méthode d'instance désigne
  - l'instance qui a reçu le message (l'instance courante), par le mot-clé `this`
  - donc, les membres de l'instance courante en les préfixant par « `this.` »
- Lorsqu'il n'y a pas d'ambiguïté, `this` est optionnel pour désigner un membre de l'instance courante

```
public class Employe {  
    private double salaire;  
    . . .  
    public void setSalaire(double unSalaire) {  
        salaire = unSalaire;  
    }  
    public double getSalaire() {  
        return salaire;  
    }  
}
```

# this explicite

this est utilisé surtout dans 2 occasions :

- pour distinguer une variable d'instance et un paramètre qui ont le même nom :

```
public void setSalaire(double salaire)
    this.salaire = salaire;
}
```

- un objet passe une référence de lui-même à un autre objet :

```
salaire = comptable.calculeSalaire(this) ;
```

# this est final !

this se comporte comme une variable final (mot-clé étudié plus loin), c'est-à-dire qu'on ne peut le modifier ; le code suivant est interdit :

```
this = valeur;
```



# Méthodes et variables de Classe



# Variables de classe

Certaines variables sont partagées par toutes les instances d'une classe. Ce sont les variables de classe (modificateur `static`)

Si une variable de classe est initialisée dans sa déclaration, cette initialisation est exécutée une seule fois quand la classe est chargée en mémoire

# Exemple de variable de classe

```
public class Employe {  
    private String nom, prenom;  
    private double salaire;  
    private static int nbEmployes = 0;  
    // Constructeur  
    public Employe(String n, String p) {  
        nom = n;  
        prenom = p;  
        nbEmployes++;  
    }  
    . . .  
}
```



# Variable de classe final

- Une variable de classe static final est constante dans tout le programme
- exemple :

```
static final double PI = 3.14;
```

- Une variable de classe static final peut ne pas être initialisée à sa déclaration mais elle doit alors recevoir sa valeur dans un bloc d'initialisation static

# Méthodes de classe

- Une méthode de classe (modificateur static en Java) exécute une action indépendante d'une instance particulière de la classe
- Une méthode de classe peut être considérée comme un message envoyé à une classe
- Exemple :

```
public static int getNbEmployes() {  
    return nbEmployes;  
}
```

# Appeler une méthode de classe

- Depuis une autre classe, on la préfixe par le nom de la classe :

```
int n = Employe.getNbEmploye();
```

- Depuis sa classe, le nom de la méthode suffit
- On peut aussi la préfixer par une instance quelconque de la classe (à éviter car cela nuit à la lisibilité : on ne voit pas que la méthode est static) :

```
int n = e1.getNbEmploye();
```

# Méthodes de classe

Comme une méthode de classe exécute une action indépendante d'une instance particulière de la classe, elle ne peut utiliser de référence à une instance courante (this)

Il est interdit d'écrire

```
static double tripleSalaire() {  
    return this.salaire * 3;  
}
```

Une méthode de classe ne peut avoir la même signature qu'une méthode d'instance

# Question

La méthode `main()` est nécessairement static.

Pourquoi ?

# Blocs d'initialisation static

- Ils permettent d'initialiser les variables static trop complexes à initialiser dans leur déclaration :

```
class UneClasse {  
    private static int[] tab = new int[25];  
    static {  
        for (int i = 0; i < 25; i++) {  
            tab[i] = -1;  
        }  
    }  
    ...  
}
```

- Ils sont exécutés une seule fois, quand la classe est chargée en mémoire

# Blocs d'initialisation non static

- Ils servent à initialiser les variables d'instance (ou toute autre initialisation)
- Ils peuvent être utiles en particulier pour les classes internes anonymes (étudiées dans un autre cours) et pour partager du code entre plusieurs constructeurs (leur code est répété par tous les constructeurs)
- La syntaxe est celle des blocs static sans le mot-clé static

```
class UneClasse {  
    private int[] tab = new int[25];  
    {  
        for (int i = 0; i < 25; i++) {  
            tab[i] = -1;  
        }  
    }  
}
```

# Représentation “à la UML”

Les classes sont représentées graphiquement sous cette forme simplifiée :

<b>Cercle</b>
private Point centre private int rayon
public Cercle(Point, int) public void setRayon(int) public int getRayon() public double surface()

<b>Cercle</b>
- Point centre - int rayon
+ Cercle(Point, int) + void setRayon(int) + int getRayon() + double surface()

(- : private, # : protected, + : public, \$ (ou souligné) : static)



# Classes enveloppes

# Enveloppes des types primitifs

- En Java certaines manipulations nécessitent de travailler avec des objets (instances de classes) et pas avec des valeurs de types primitifs
- Le paquetage `java.lang` fournit des classes pour envelopper les types primitifs : `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, `Character`
- Attention, les instances de ces classes ne sont pas modifiables (idem `String`)

# Exemple de la classe Integer

- $\text{int} \rightarrow \text{Integer}$  :

`new Integer(int i)`

- $\text{Integer} \rightarrow \text{int}$  :

`int intValue()`

- $\text{String} \rightarrow \text{Integer}$  :

`static Integer valueOf(String ch [,int base])`

- $\text{Integer} \rightarrow \text{String}$  :

`String toString()`

# AutoBoxing

- Le « autoboxing » (mise en boîte) automatise le passage des types primitifs vers les classes qui les enveloppent
- Cette mise en boîte automatique a été introduite par la version 5 du JDK
- L'opération inverse s'appelle « unboxing »
- Le code précédent peut maintenant s'écrire :

```
liste.add(89) ;
```

```
int i = liste.get(n) ;
```

- Mais il fait la même chose !!

# Exemples de boxing/unboxing

- Integer a = 89;  
a++;  
int i = a;
- Integer b = new Integer(1); // unboxing suivi de boxing  
b = b + 2;
- Double d = 56.9;  
d = d / 56.9;
- Transformer un int en Long :  
Long l = (long)i;
- Attention, une tentative de unboxing avec la valeur null va lancer une NullPointerException

# Jeu 1 ! [enfin !!!!] : compilo ?

```
class TapeDeck {
    boolean canRecord = false;
    void playTape() {
        System.out.println("tape playing");
    }
    void recordTape() {
        System.out.println("tape recording");
    }
}

class TapeDeckTestDrive {
    public static void main(String [] args) {
        t.canRecord = true;
        t.playTape();
        if (t.canRecord == true) {
            t.recordTape();
        }
    }
}
```

# Jeu 2 : compilo ?

```
class DVDPlayer {
    boolean canRecord = false;
    void recordDVD() {
        System.out.println("DVD recording");
    }
}

class DVDPlayerTestDrive {
    public static void main(String [] args) {
        DVDPlayer d = new DVDPlayer();
        d.canRecord = true;
        d.playDVD();
        if (d.canRecord == true) {
            d.recordDVD();
        }
    }
}
```

# Jeu 3 : chamboule tout

```
void jouerCymbale() {  
    System.out.println("ding ding  
da-ding");  
}
```

```
b.jouerCaisseClaire()
```

```
Batterie b = new Batterie();
```

```
class BatterieTestDrive {
```

```
b.caisseClaire = false;
```

```
    if (b.caisseClaire == true) {  
        b.jouerCaisseClaire();  
    }
```

```
void jouerCaisseClaire() {  
    System.out.println("bang  
bang da-bang");  
}
```

```
boolean cymbales = true;  
boolean caisseClaire = true;
```

```
public static void main(String... args) {
```

```
b.jouerCymbale();
```

```
class Batterie {
```



# Exercice de programmation

Créer une classe Etudiant en considérant les données suivantes :

- Un étudiant a un nom, un prénom, une année de naissance et un identifiant automatiquement créé à la génération
- Nous devons permettre l'accès en lecture à chacune de ses informations (attention les champs doivent rester privés)
- Nous devons permettre l'accès en écriture à tous les champs sauf l'identifiant
- Nous devons développer une méthode permettant de retourner une String avec toutes les informations de l'étudiant

# Exercice avancé

Nous voulons créer une classe *DieuJava* qui n'a aucune variable d'instance particulière. La seule contrainte que nous voulons définir est qu'un utilisateur ne puisse pas créer deux instances différentes de cette classe. Imaginez un mécanisme obligeant le développeur possédant votre classe à respecter cette contrainte.

Indice : une variable de classe bien pensée et des constructeurs inaccessibles



# Les tableaux

Mise à niveau Java  
Y. Boichut & F. Moal



# Les tableaux - des objets

En Java, les tableaux sont considérés comme des objets :

- les variables de type *tableau* stockent comme valeur une adresse
- les tableaux sont créés par l'opérateur **new**
- ils possèdent une variable d'instance **length** (final)
- ils héritent des méthodes d'instance de la classe **Object**

# Utilisation des tableaux

- Affectation des cases - tableau indexé de 0 à sa taille -1
  - `notes[0]=1;`
- Accès à la taille déclarée
  - `notes.length`
- Déclaration dans la signature d'une méthode - identique à la déclaration dans un bloc
  - `double[] m(int[] monTableau)`

# Afficher les éléments d'un tableau

- La méthode `toString()` prédéfinie ne suffira malheureusement pas
  - Par exemple : `System.out.println((new double[]{1,2,3}).toString())` n'affichera pas `[1, 2, 3]`
- Par contre, utilisation de la méthode statique de la classe `Arrays`
  - `Arrays.toString((new double[]{1,2,3}))` aura l'effet attendu
- Parcours par une boucle
  - `for (double x : notes) {System.out.println(x+" ");}`
  - ou `for (int i = 0; i < notes.length; i++) {System.out.println(notes[i]+" ");}`
  - ou ...

# D'autres méthodes/fonctions de la classe Arrays

- *Arrays.equals(t1,t2)* retourne vrai si les deux tableaux contiennent des valeurs identiques à chaque indice
  - Evidemment comme pour *toString()*, *t1.equals(t2)* ne retourne vrai que si t1 et t2 référencent la même adresse
- *Arrays.sort(t)* trie un tableau
- *Arrays.binarySearch(t,e)* effectue une recherche dichotomique de l'élément e dans t
- *Arrays.fill(t,v)* remplit le tableau t avec la valeur v
- Plus d'informations à <https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html>

# Tableaux à plusieurs dimensions

- Déclaration

```
double[][] notes;
```

- Chaque élément du tableau notes contient une référence vers un tableau de *doubles*

- *Création*

- *notes = new notes[5][6]; /\*matrice à dimensions fixes\*/*
- *notes = new notes[5][]; /\*matrice avec une dimension variable\*/*

- *Initialisation*

- *double[][] notes = {{1,2,3}, {4,5}};*
- *notes = new double[][]{{1,2,3}, {4,5}}*

- *Affectation*

- *notes[0][2] = 3;*



# Quelques méthodes spécifiques de la classe Arrays

- `Arrays.deepEquals(t1,t2)` : permet de tester l'égalité sémantique de deux tableaux à plusieurs dimensions
- `Arrays.deepToString(t)` : permet de construire une chaîne de caractère représentant le contenu d'un tableau à plusieurs dimensions

# Jeu 1 : compilo

```
class Books {
    String title;
    String author;
}

class BooksTestDrive {
    public static void main(String [] args) {
        Books [] myBooks = new Books[3];
        int x = 0;
        myBooks[0].title = "The Grapes of Java";
        myBooks[1].title = "The Java Gatsby";
        myBooks[2].title = "The Java Cookbook";
        myBooks[0].author = "bob";
        myBooks[1].author = "sue";
        myBooks[2].author = "ian";
        while (x < 3) {
            System.out.print(myBooks[x].title);
            System.out.print(" by ");
            System.out.println(myBooks[x].author);
            x = x + 1;
        }
    }
}
```

# Exercice - plateau démineur

Ecrire une classe *Plateau* avec un champ *int[ ][ ] plateau*. Le constructeur prendra deux arguments i.e. la largeur et la longueur du plateau.

- Définir la méthode *initialiser(int x)* permettant d'initialiser correctement le plateau i.e. *plateau[i][j]* sera un entier spécifiant le nombre de bombes dans son voisinage direct ou -1 si c'est une bombe. Cette méthode devra tirer aléatoirement le placement des x bombes. A chaque placement, nous incrémenterons le voisinage direct. Après le placement de la dernière bombe, nous aurons alors un plateau correct.
- Définir un affichage terminal visuellement acceptable de notre plateau.



# Strings

Mise à niveau Java  
Y. Boichut & F. Moal



# Strings

- 2 types de strings
  - *String* pour les chaînes constantes
  - *StringBuilder* ou *StringBuffer* pour les chaînes variables

# L'objet String

- Affectation
  - `chaine = "ma string";`
- En Java, les deux variables ci-dessous vont référencer le même objet
  - `chaine1 = "maString";`
  - `chaine2 = "maString";`

*Ainsi, `chaine1 == chaine2`, mais pas pour la bonne raison !!*

- Par contre ça n'aurait pas été le cas si nous avions eu : `chaine2 = new String("maString");`

# Concaténation

- opérateur de concaténation '+'

`int x=5;`

`s = "Valeur de x" + x;`

- Types primitifs sont gérés par le compilateur
- pour les instances de classes autres, la méthode `toString()` est automatique appelée

# Egalité de Strings

- La méthode *equals* teste si deux instances de String ont sémantiquement la même valeur
  - ("salut "+"la compagnie").equals("salut la compagnie") => TRUE
  - String c1 = "salut ";
  - (c1+"la compagnie").equals("salut la compagnie") => TRUE
  - (c1+"la compagnie") == "salut la compagnie" => FALSE
- *equalsIgnoreCase()* est un test d'égalité en ne tenant pas compte des minuscules et des majuscules



# Comparaison lexicographique

- `c1.compareTo(c2)` renvoie le signe de “`c1-c2`”
  - 0 : égalité entre `c1` et `c2`
  - `<0` : `c2` suit `c1` dans l'ordre lexicographique (`c1 < c2`)
  - `>0` : `c2` précède `c1` dans l'ordre lexicographique (`c1 > c2`)
- `c1.compareToIgnoreCase(c2)` fait la même chose en ignorant les majuscules et minuscules

# Quelques méthodes/fonctions de la classe String

- `int indexOf(String s), int indexOf(String s, int d)`
- `int lastIndexOf(String s), int lastIndexOf(String s, int d)`
- `String substring(int d, int f), String substring(int d)`
- `boolean startsWith(String s), boolean startWith(String s, int i), boolean endsWith(String s)`
- `String trim()`
- `String toUpperCase()`
- `String toLowerCase()`
- `String valueOf(<Type primitif> t)` - méthode statique

# Exercice

Soit une classe `Url` qui possède deux champs - `String` `protocole` et `String[]` `adresse`. Le tableau contient chaque composant de l'adresse. Par exemple, pour l'adresse <http://www.je-comprends.org/que/dalle>, le tableau contiendra dans la case 0 : "[www.je-comprends.org](http://www.je-comprends.org/que/dalle)", dans la case 1 : "que"... Le champ `protocole` sera alors initialisé avec la valeur "http".

- Ecrire la classe et ses constructeurs de bases
- Ecrire la méthode `void initialiseURL(String url)` qui permettra d'initialiser à partir d'une URL (que nous supposerons correcte) de remplir les champs spécifiques

# Strings modifiables

- *StringBuffer* / *StringBuilder* : le dernier a été ajouté dans la JDK 5 et reste stable dans un environnement concurrent (contrairement au premier)
- *append*
- *insert*
- *replace*
- *charAt*
- *substring*
- *reverse*
- Pas de *equals* définit dans ces classes

=> construction par calcul (boucle), puis récupération de la chaîne finale :  
`toString()`



# Les exceptions

Mise à niveau Java  
Y. Boichut & F. Moal



# Erreurs / Exceptions

- On utilise les erreurs / exceptions pour traiter un fonctionnement anormal d'une partie du code (provoqué par une erreur ou un cas exceptionnel (le fameux "normalement, on ne devrait jamais arriver ici"))
- En Java, une erreur provoque la création d'un objet *Exception* ou *Error*

# Localisation du traitement exceptionnel

- Le traitement des événements exceptionnels se fait dans une zone spéciale appelée bloc *"catch"*

```
try {  
  
    recuperationDesDevoirs.add(etudiantBrillant.rendreDevoir());  
  
}  
  
catch (MonChienAMangeMonOrdiException etudiant) {...  
  
    System.err.println("Chien du futur");  
  
}
```

# Bloc try-catch

```
try {  
    ...  
}  
  
catch (WTFException1 e) {...}  
catch (WTFException2 e) {...}  
catch (WTFException3 | WTFException4 e) {...}
```

- Multi-catch possible depuis JDK 7 mais les classes d'exceptions concernées ne doivent pas être dans la même “descendance” familiale
  - `catch(NumberFormatException | Exception e){...}` n'est pas compilable  
“Types in multi-catch must be disjoint”
- Importance de l'ordre des clauses “catch” - les plus générales à la fin



# Dialecte

- **Lever** une exception : une exception est créée puis remontée à la méthode appelante
- **Traiter** une exception : l'exception est capturée et traitée dans la méthode courante
- **Remonter** une exception : l'erreur remonte vers la méthode appelante et n'est pas gérée dans la méthode courante
- **Exceptions contrôlées** : EOFException ou tout autre exception créée par le développeur
- **Exceptions non contrôlées** : OutOfBoundException, ...

# Précisions pour les clauses “catch”

Les exceptions contrôlées spécifiées dans les clauses “catch” doivent être susceptibles d’être levées dans les instructions du bloc “try”

“Exception Blabla never thrown in the corresponding try block”

Ce qui n’est absolument pas le cas pour les exceptions non contrôlées

# Scénarios : exception levée dans un bloc try

Les instructions du bloc try suivant l'instruction responsable de la levée ne sont pas exécutées

- Si au moins une clause catch capture l'exception
  - la première clause catch appropriée est exécutée
  - le traitement reprend après le bloc try-catch
- Sinon
  - la méthode courante se termine immédiatement
  - l'exception est levée à la méthode appelante

# Scénarios : aucune exception levée dans un bloc *try*

- le déroulement du bloc *try* s'effectue normalement et intégralement
- le programme se poursuit ensuite après les clauses *catch*

# Scénarios : exception jamais traitée

- L'exception remonte jusqu'au *main*
- Le programme s'arrête immédiatement
- Le message associé à l'exception levée est affiché avec une description des méthodes traversées par l'exception
- Dans le cas d'un multi-thread, uniquement le thread concerné meurt

# Que fait on dans un bloc catch ?

- Traduire l'exception levée en une autre exception de plus haut niveau dans le cadre d'une application multi-couches (cf IHM)
- Retourner une valeur particulière - exemple : retour de -1 lorsqu'on cherche l'indice d'une valeur dans un tableau
- Faire un traitement alternatif

# Traitement d'une exception

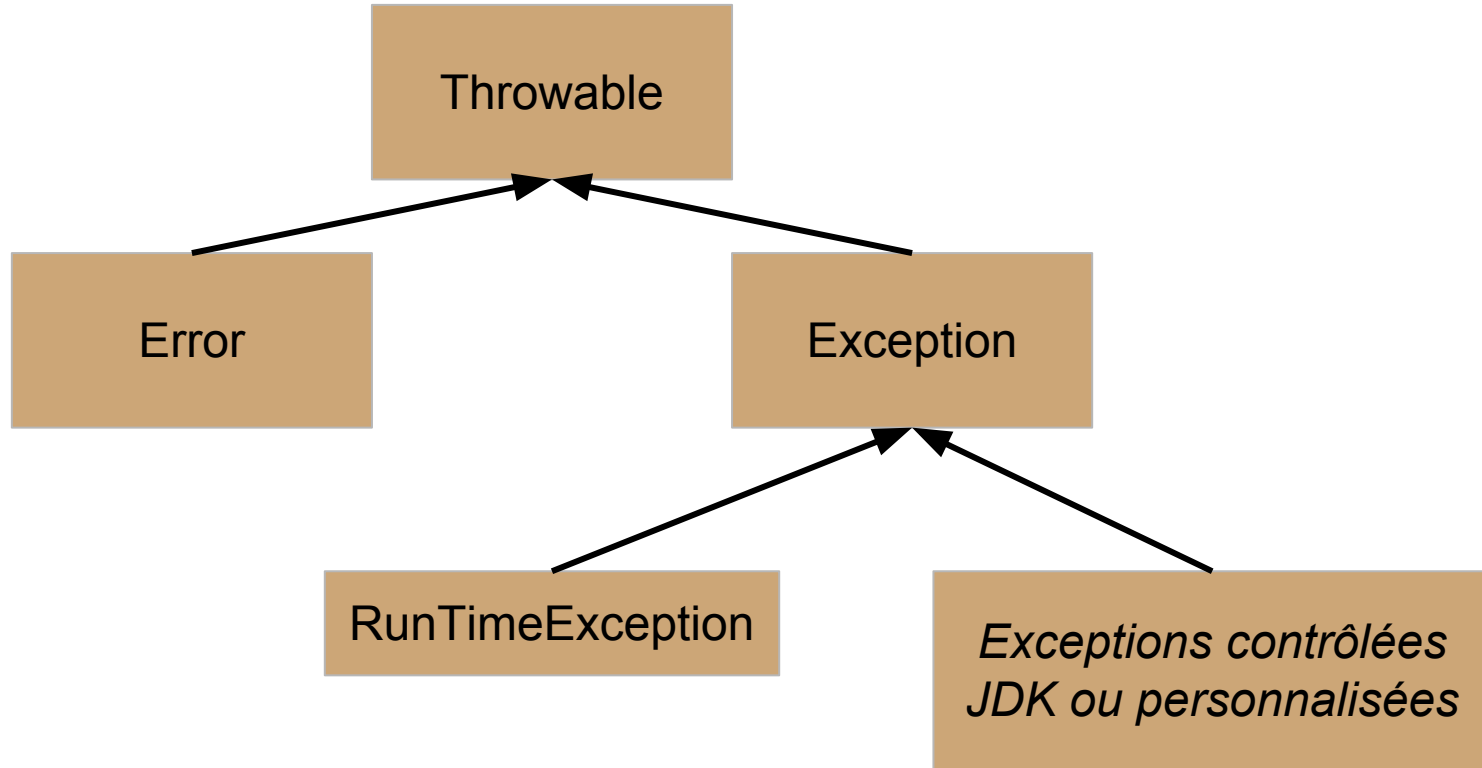
1. Traiter complètement l'exception dans la méthode concernée
2. Laisser remonter l'exception
3. Traiter partiellement l'exception, puis lever une nouvelle exception

# Pourquoi laisser remonter une exception ?

- Plus une méthode est éloignée du *main* moins elle a une connaissance globale de l'application
- Elle peut donc laisser une méthode mieux placée pour traiter l'exception



# Hiérarchie des exceptions / errors



# Hiérarchie des exceptions / errors

- RuntimeException
  - NullPointerException
  - ClassCastException
  - ArrayIndexOutOfBoundsException
- Exceptions contrôlées de la JDK
  - FileNotFoundException
  - EOFException
  - ClassNotFoundException

# Méthode susceptible de lever une exception

- Utilisation du mot clé *throws* dans le profil de la méthode

```
public Devoir rendreDevoir() throws MonChienAMangeMonOrdiException
```

# Créer ses exceptions

1. Créer une classe étendant la classe Exception
  - par convention le nom de la classe doit se terminer par Exception
2. Créer deux constructeurs
  - Un sans paramètre
  - Un avec une chaîne de caractères en paramètres
3. Les deux constructeurs appelleront systématiquement le constructeur de la classe mère avec l'instruction *super*

# Lever une exception

- Utilisation de l'instruction *throw*

```
throw new MonChienAMangeMonOrdiException();
```

- La méthode concernée doit être déclarée comme susceptible de lever cette même exception

# Mise en application

Live démo - développement d'une application d'un prof collectant les devoirs de ses élèves

# Savoir lire un message d'erreur.....

- Le message commence par le message d'erreur associé à l'erreur qui a provoqué l'affichage
- On a ensuite la pile des méthodes traversées en attente de résultat depuis la méthode *main* pour arriver à l'erreur
- Les méthodes les plus récemment invoquées sont en premier, la méthode *main* est en dernier
  - A chaque étape on a le numéro de la ligne en cause



# Héritage et polymorphisme

Mise à niveau Java  
Y. Boichut & F. Moal





# Notion d'héritage

- L'héritage permet d'écrire qu'une classe **B** se comporte comme une classe **A** mais avec quelques différences
- Uniquement besoin du code compilé de la classe **A**
- Code source de **B** ne comporte que ce qui change par rapport au code de **A**
  - Ajouter de nouvelles variables
  - Ajouter des nouvelles méthodes
  - Modifier des méthodes de **A** - redéfinition

# Vocabulaire

- La classe B hérite de la classe A
  - B est une classe fille ou sous-classe de A
  - A est la classe mère ou super-classe

# L'héritage

- Particularisation - généralisation
  - Un polygone est une figure géométrique mais une figure géométrique particulière
  - la notion de figure géométrique est une généralisation de la notion de polygone
- Une classe fille offre de nouveaux services ou enrichit les services rendus par la classe mère
  - Exemple : Rectangle / RectangleColore
- Chaque langage objet a ses particularités
  - C++ et Eiffel supportent l'héritage multiple alors que ce n'est pas le cas pour C# et Java (encore que....)

# Héritage en Java

- Chaque classe a **une et une** seule classe mère
- Le mot “**extends**” indique la classe mère et définit ainsi l’héritage
  - `class RectangleColore extends Rectangle`
- Tout objet en Java hérite de la classe **Object**

# Ce que peut faire une classe fille

- La classe qui hérite peut
  - ajouter des champs, des méthodes et des constructeurs
  - redéfinir des méthodes (même signature)
  - surcharger des méthodes (même nom mais pas même signature)
- La classe qui hérite ne peut retirer aucun champ ou aucune méthode de la classe mère

# Principe important

- Si “B extends A” alors *tout B est un A*
- B est un sous-type de A
  - On peut stocker une valeur de type B dans une variable de type A
  - Exemple : `A a = new B();`
  - L'inverse n'est pas possible sauf en forçant par Cast (pouvant échouer)

# Quelques compléments sur les constructeurs

- La première instruction d'un constructeur peut être un appel
  - à un constructeur de la classe mère (pour initialiser des champs de la classe mère)
    - `super(...)`
  - à un autre constructeur de la classe courante
    - `this(...)`
- Impossible de placer ces instructions ailleurs qu'en première instruction d'un constructeur de la classe courante

# Quelques choses à savoir sur les constructeurs

- Si aucun constructeur n'est spécifié pour une classe donnée
  - alors il existe un constructeur par défaut (automatique généré à la compilation)
- Si un constructeur avec paramètres est spécifié et qu'un constructeur par défaut ne l'est pas
  - alors aucun constructeur par défaut n'est généré
- Si un constructeur avec paramètres est spécifié dans une classe mère
  - alors les filles doivent également implanter un constructeur avec ces paramètres



# Exercice

```
public class Personnage {  
    long pointVie;  
    public Personnage(long pointVie) {  
        this.pointVie = pointVie;  
    }  
}
```

Créez la classe Troll qui héritera de la classe **Personnage**

# Exercice

Plus de constructeurs  
par défaut

```
public class Troll extends Personnage {  
    public Troll(long pointVie) {  
        super(pointVie);  
    }  
}
```

Appels implicite de  
Objet()

# Accessibilité

- `private`
- `package` (protection par défaut)
- `protected` (fortement déconseillée - encapsulation d'une classe mère)
- `public`

## Bonnes pratiques

- Utiliser uniquement `private` / `public` et getters/setters

# La classe Object

- En Java, `java.lang.Object` est la classe **racine** de l'arbre d'héritage
- Object n'a
  - ni variables de classes
  - ni variables d'instances
- Object a des méthodes/fonctions
  - `String toString()` : description de l'objet sous forme de caractère - classe & adresse par défaut
  - `boolean equals(Object o)` : égalité sémantique - égalité d'adresse par défaut
  - `int hashCode()` : valeur entière représentant l'objet - renvoie la valeur hexa de l'adresse mémoire par défaut
  - + quelques autres *dark* functions

# Redéfinitions de méthodes / fonctions

- Annotation @Override

```
@Override  
public boolean equals(Object x) {  
    ...  
}
```

- Annotation INDISPENSABLE pour repérer les erreurs de saisie (nom différent ou profil pas identique) - Assurance d'une redéfinition effective

# Bon chasseur ou mauvais chasseur ?

- Ne pas confondre **redéfinition** et **surcharge**
- Une méthode **redéfinit** une méthode héritée quand elle a la **même signature** que l'autre méthode
- Une méthode **surcharge** une méthode héritée (ou définie dans la classe) quand elle a le **même nom** mais **pas le même profil** que l'autre méthode

# Exemple

Démo avec une super classe Nombre

# Polymorphisme

- En Java pas 36 formes de polymorphisme
- Héritage multiple non autorisé (**attention à Java 8**)
- Implémentations multiples *d'interfaces* autorisées (**plus tard**)
- Le polymorphisme est le fait qu'une même expression peut correspondre à différents appels de méthodes
  - *late binding*

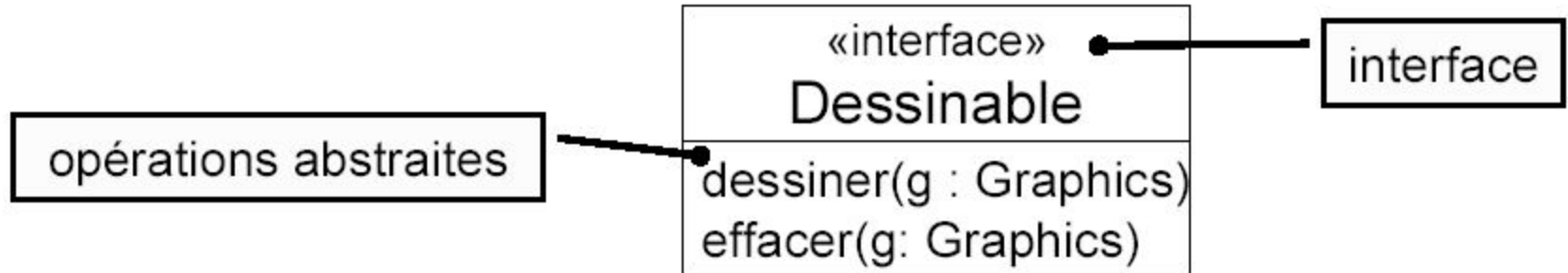


# Cast ou Transtypage

- *Caster* un objet = forcer le type de l'objet qui n'est pas le type déclaré ou réel de l'objet
- Les seuls cast autorisés sont ceux entre classes mères et classes filles
  - UPcast - souvent implicite
  - DOWNcast - doit être explicite et peut provoquer des erreurs à l'exécution
- Cas particulier - démo avec les double-int / Double-Integer

# Interfaces

- Une interface est une collection d'opérations utilisée pour spécifier un service offert par une classe
- C'est un contrat spécifié par une classe par le mot clé implements pour dire que l'on respecte cette interface
- Exemple :



# Interfaces

- Toutes les méthodes sont abstraites (pas de corps)
- 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

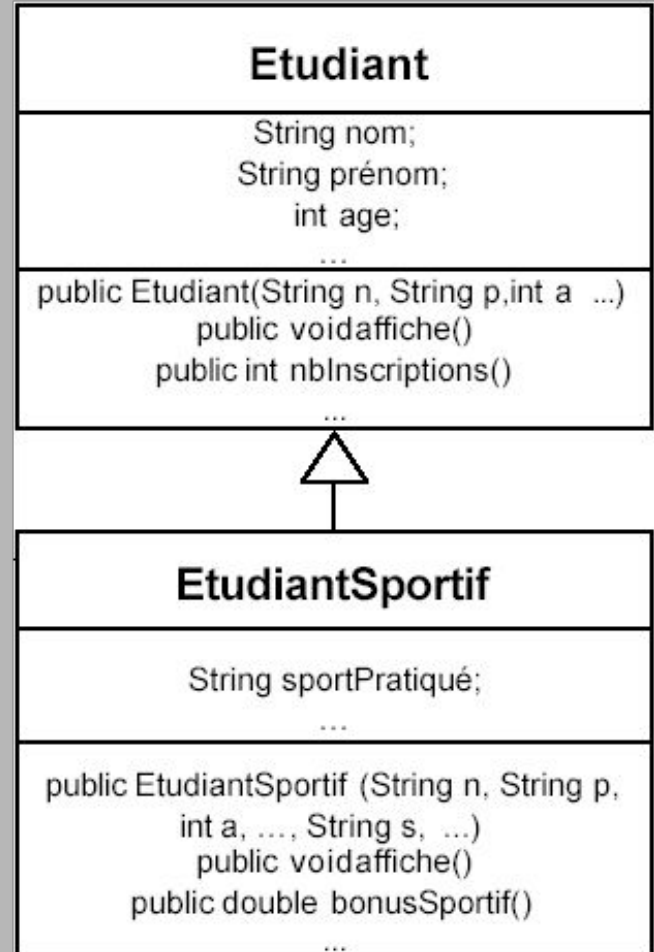
```
public interface Dessinable {  
    public void dessiner(Graphics g);  
    void effacer(Graphics g);  
}
```

# Interface : exemple

```
interface EstType {  
    void afficherType();  
}  
  
class Personne implements EstType, Parle {  
    public void afficherType() {  
        System.out.println("Je suis une personne ");  
    }  
    public void disBonjour() {  
        System.out.println("Bonjour !");  
    }  
}  
  
class Voiture implements EstType {  
    public void afficherType() {  
        System.out.println("Je suis une voiture "); }  
}
```

# Question : compile ? Run ?

```
EtudiantSportif es;  
es = new EtudiantSportif("DUPONT",  
    "fred",25,...,"Badminton",...);  
Etudiant e;  
e = es; // upcasting  
e.affiche();  
es.affiche();  
e.nbInscriptions();  
es.nbInscriptions();  
es.bonusSportif();  
e.bonusSportif();
```



# Jeu 1

```
class A {
    int ivar = 7;
    void m1() {
        System.out.print("A m1, ");
    }
    void m2() {
        System.out.print("A m2, ");
    }
    void m3() {
        System.out.print("A m3, ");
    }
}

class B extends A {
    void m1() {
        System.out.print("B m1, ");
    }
}
```

```
class C extends B {
    void m3() {
        System.out.print("C m3, "+(ivar + 6));
    }
}

public class Melange {
    public static void main(String [] args) {
        A a = new A();
        B b = new B();
        C c = new C();
        A a2 = new C();

        ! code ici !
    }
}
```

# Match ?

b.m1();

c.m2();

a.m3();

----

c.m1();

c.m2();

c.m3();

----

a.m1();

b.m2();

c.m3();

----

a2.m1();

a2.m2();

a2.m3();

?

A m1, A m2, C m3, 6

B m1, A m2, A m3,

A m1, B m2, A m3,

B m1, A m2, C m3, 13

B m1, C m2, A m3,

B m1, A m2, C m3, 6

A m1, A m2, C m3, 13

# jeu 2 : trouver le(s) A/ B/ qui marche...

```
public class MonstreTestDrive {  
    public static void main(String [] args) {  
        Monstre [] ma = new Monstre[3];  
        ma[0] = new Vampire();  
        ma[1] = new Dragon();  
        ma[2] = new Monstre();  
        for(int x = 0; x < ma.length; x++) {  
            ma[x].fairepeur(x);  
        }  
    }  
}
```

```
class Monstre {  
    // A  
}  
  
class Vampire extends Monstre {  
    // B  
}  
  
class Dragon extends Monstre {  
    boolean fairepeur(int degree) {  
        System.out.println("souffler du feu");  
        return true;  
    }  
}
```

```
# java MonstreTestDrive  
mordre ?  
souffler du feu  
arrrrgh
```



# ça marche ?

```
//A
boolean fairepeur(int d) {
    System.out.println("arrrgh");
    return true;
}

// B
boolean fairepeur(int x) {
    System.out.println("mordre ?");
    return false;
}
```

```
// A
boolean fairepeur(int x) {
    System.out.println("arrrgh");
    return true;
}

// B
int fairepeur(int f) {
    System.out.println("mordre ?");
    return 1;
}
```

# ça marche ?

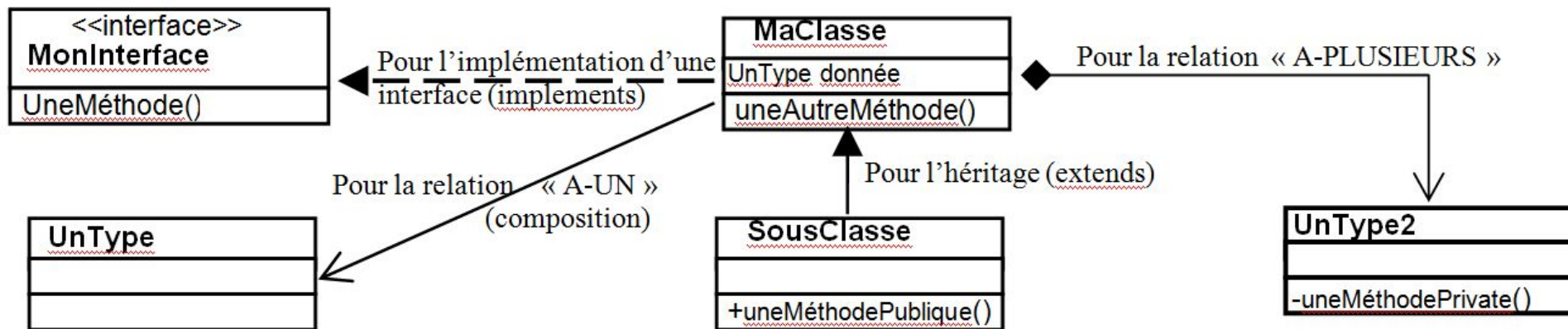
```
// A
boolean fairepeur(int x) {
    System.out.println("arrrgh");
    return false;
}
```

```
// B
boolean effrayer(int x) {
    System.out.println("mordre ?");
    return true;
}
```

```
// A
boolean fairepeur(int z) {
    System.out.println("arrrgh");
    return true;
}
```

```
// B
boolean fairepeur(byte b) {
    System.out.println("mordre ?");
    return true;
}
```

# Dessin !



# A vous de jouer 4 :

```
public interface Foo { }  
public class Bar implements Foo { }
```

```
public interface Vinn { }  
public abstract class Vout implements Vinn { }
```

```
public abstract class Muffie implements Whuffie { }  
public class Fluffie extends Muffie { }  
public interface Whuffie { }
```

```
public class Zoop { }  
public class Boop extends Zoop { }  
public class Goop extends Boop { }
```

```
public class Gamma extends Delta  
implements Epsilon { }  
public interface Epsilon { }  
public interface Beta { }  
public class Alpha extends Gamma  
implements Beta { }  
public class Delta { }
```

# Jeu 5

Un robot qui avance

Un robot qui roule

un robot qui marche

# Jeu 5 - évolution

Un robot qui avance

Ce robot peut rouler quand le terrain est plat

Ce robot marche quand le terrain est accidenté

# Jeu 6 sur machine

Passionné de zoologie, vous avez décidé de construire une application Java permettant de représenter différents types d'animaux. Vous avez commencé par vos animaux préférés, les canards, en créant une interface Canard possédant deux méthodes :

- void cancaner() qui permet de faire caqueter le canard
- dandiner(double distance) qui permet de faire marcher le canard sur une certaine distance

Vous avez ensuite implanté plusieurs réalisations de cette interface : la classe Colvert et la classe Mandarin

# Jeu 6 sur machine

En avançant dans la construction de votre application, vous avez défini une nouvelle interface Animal plus générale avec les méthodes suivantes :

- void faireDuBruit()
- avance(double distance) qui permet de faire bouger l'animal en question

Ecrire une classe Chien et Dindon qui implémentent cette interface.



# Jeu 6 sur machine

Vous vouliez pouvoir gérer des collections complètes d'animaux variés en récupérant le code de vos canards. Malheureusement, le disque dur de votre vieil ordinateur vous a lâché et vous ne disposez plus que du bytecode de l'interface Canard et de ses réalisations. Impossible donc de modifier le code source ces classes et vous ne pouvez pas les réécrire, les caquètements des canards vous ayant demandé trop de temps à implanter !

Proposez une solution permettant de considérer vos canards (Colvert, Mandarin) comme des objets de type Animal, pour pouvoir les insérer dans un tableau d'Animal ; écrire un main faisant avancer un troupeau d'animaux comprenant 2 chiens, un colvert, un didon et un mandarin.