Modèle d'exécution Java

Rémi Forax
forax@univ-mlv.fr

Plan

- Type primitifs
- Structure de controle, variable locale
- Paquetage
- Classe/constructeur et statique
- Boxing/unboxing
- Exception

Les Types et Java

- Philosophiquement, Java sépare les types primitifs des types Objets.
  - Les types primitifs :
    Ex: boolean, int, double
  - Les types objet
    Ex: String, int[], Point

- Les types primitifs sont manipulés par leur valeur, les types objets par référence.

Les types primitifs

Il existe 8 types primitifs (et c'est tout) :

- Valeur booléen : boolean (true/false)
- Valeur numérique entière signée :
  byte(8 bits)/short(16)/int(32)/long(64)
- Valeur numérique flottante (IEEE 754)
  float(32 bits)/double(64)
- Charactère unicode :
  char(16 bits)

Les valeurs booléennes

- Correspond aux valeurs qui peuvent être testé par un **if**.

```
...
public void test(int i,int j) {
  boolean v=(i==3 && j==4);
  if (v)
    doSomething();
}
...
```

- Deux valeurs possibles :
  true (vrai) et false (faux)

Les caractères

- Les caractères sont codés en Unicode sur **16 bits** (non signés).
- Les 128 premières valeurs sont identiques au code ASCII.
- Un caractère se définie entre quotes ' ' Ex: 'A' ou 'a'
- Valeurs remarquables
  - '\n', '\t', '\r' etc.
  - '\uxxxx' (xxxx en hexa)

Convertion de caractères

- Les caractères d'un fichier ou reçu par le réseau sont rarement en unicode
  - Il faudra donc convertir les caractère 8 bits (charset particulier) en unicode (16 bits)
- Java par défaut utilise la convertion de la plateforme (source de bcp de bugs)
  - Unix (dépend ISO-Latin1, UTF8, UTF16)
  - Windows (Windows Latin1)
  - Mac-OS (...)

Les valeurs entières signés

- Les valeurs sont **signées** :
  ex: byte va de -128 à 127.
- La VM ne connait que les entiers 32/64 bits :
  byte/short et char sont sujets à la promotion entière

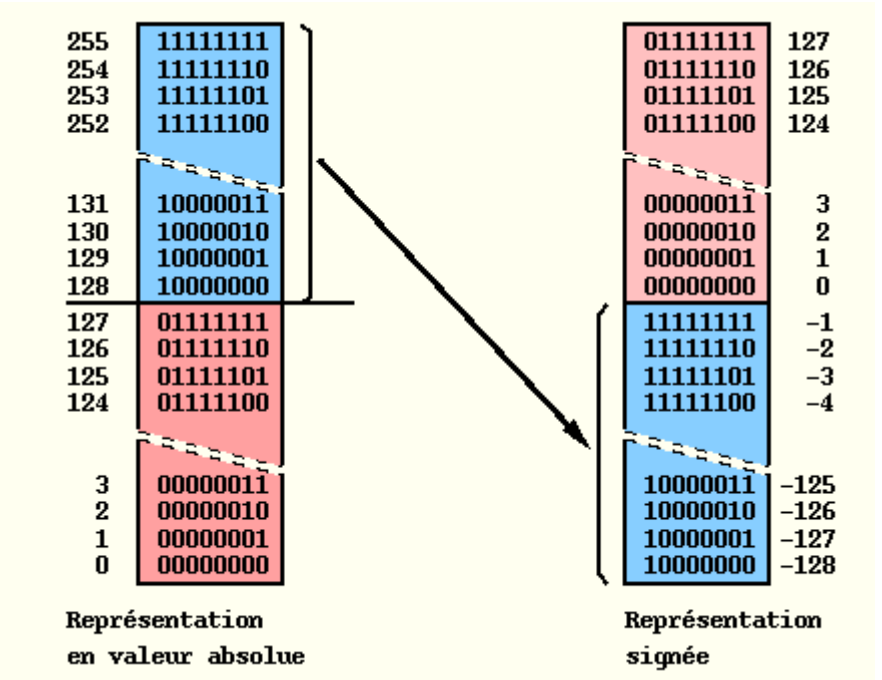
```
...
public void test() {
  short s=1;
  s=s+s;  // erreur de compilation
}
...
```

Le résultat est un int

Les valeurs entières signés (2)

- Valeur signée et représentation en complément à 2.

- La *conversion* préserve la valeur.




```
...
public void test() {
  byte b=-4;   // 11111100
  short s=b;   // 11111111 11111100
  System.out.println(s); // -4

  int v=b & 0xFF; // 00000000 00000000 00000000 11111100
}
...
```

Utilise un masque pour préserver la représentation

Les valeurs flottantes

- Utilise la norme IEEE 754 pour **représenter** les valeurs à virgule flottante
- Calcul sécurisé :
  - Existe +0.0 et -0.0
  - +Infinity et -Infinity
  - NaN (Not A Number)

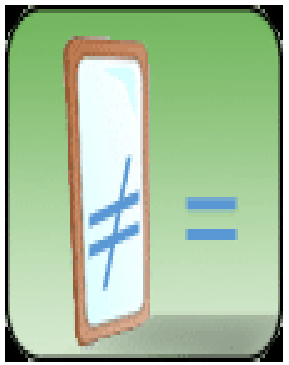
```
double[] values={ 0.0, -0.0};

for(double v:values) {
  System.out.println(3/v);
  System.out.println(v/0.0);
}
```

```
Infinity
NaN
-Infinity
NaN
```

- 3.0 est un double (64bits),
  3.0f (ou 3.0F) est un float (32 bits)

Infini et NaN

- La norme IEEE 754 introduit trois valeurs particulieres par types de flottants
  - +Infinity est le résultat de *i*/0 avec *i* positif
  - -Infinity est le résultat de *i*/0 avec *i* négatif
  - NaN est le résultat de 0/0
    - x == x est faux si x vaut Double.NaN
    - Donc on doit tester NaN avec Float.isNaN() ou Double.isNaN()



Calcul flottant

- IEEE 754, pour chaque opération +, −, *, /, √
  - On calcul le résultat précis infinie
  - On **arrondie** au nombre représentable le plus proche
- Attention à l'utilisation des flottants dans les boucles

```
for(double v=0.0;v!=1.0;v=v+1.0) {
  System.out.println(v);
}
// aie! boucle infinie
// utilisé plutot '<'
```

Strictfp

- **strictfp** : indique que les calculs effectués seront reproductibles aux bits près sur les flottants quelquesoit l'environnement

```
public class Keyword {
  public strictfp double max(double d,double d2) {
    ...
  }
}
```

- **stricfp** induit un cout lors de l'exécution

Attention aux arrondies !

- Attention les fottants sont une approximation des réels
  - 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 != 1.0
  - 0.1f != 0.1d
- **"0.1f" != 0.1** valeur après convertion =
  - 0.100000001490116119384765625
- **"0.1d" != 0.1** valeur après convertion =
  - 0.1000000000000000055511151231…

Calcul (suite)

- L'associativité ne marche pas :
  - (1.0f + 3.0e−8f) + 3.0e−8f = 1.0f
  - 1.0f + (3.0e−8f + 3.0e−8f) = 1.0000001f

- Les $, £, ., € ne doivent pas être stockées en utilisant des flottants, on utilise :
  - des **int** ou des **long** répésentant des centimes
  - ou **java.math.BigDecimal**

Flottant et String

- La convertion de/vers une String dépend du format (float/double) :
  - **Float.toString(0.1f) = "0.1"**
  - **Double.toString(0.1f) = "0.10000000149011612"**
  - **Double.toString(0.1d) = "0.1"**
    - float vers String utilise les 24 premiers bits
    - double vers String utilise les 53 premiers bits

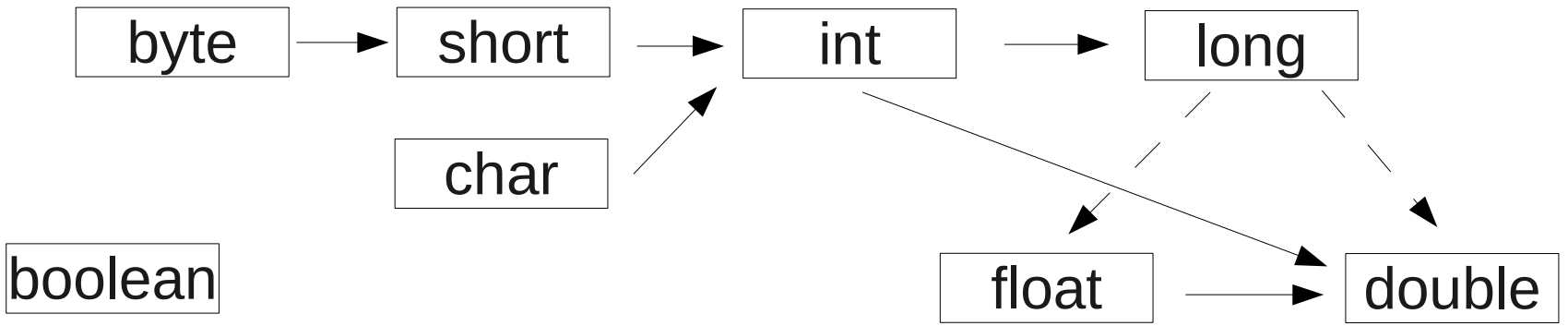
Conversion de types primitifs

- Conversion automatique :

```
byte b=3;          // ok car -128<=3<=127
int i=b;

float f=i;         // ok mais peut-être perte

double d=2.0
int j=d;         // erreur de compilation
int j2=(int)d; // ok
```

Conversion sans perte

Conversion avec perte



Méthode

- En Java, il est impossible de définir du code hors d'une méthode.
- Une méthode est séparée en 2 parties :
  - La signature (types des paramètres, type de retour)
  - Le code de la méthode
- Le code d'une méthode est constitué de différents blocs imbriqués. Chaque bloc définit et utilise des variables locales.

Structures de contrôle

- Java possède quasiment les mêmes structures de contrôle que le C.

- Différences :
  - pas de **goto**
  - **switch**(enum)
  - **for** spécial (“foreach”)
  - **break** ou **continue** avec label.

Structure de test

- Test Conditionnel booléen :

```
if (args.length==2)
  doSomething();
else
  doSomethingElse();
```

- Test Multiple numérique (ou valeur d'enum):

```
switch(args.length) {
  case 0:
    return;
  case 2:
    doSomething();
    break;
  default:
    doSomethingElse();
}
```

```
switch(threadState) {
  case NEW:
    runIt();
    break;
  case RUNNABLE:
    break;
  case BLOCKED:
    unblock();
}
```

Structure de boucle

- Boucle tant que condition booléenne vrai:
  while(*condition*){...}
  do{...}**while**(*condition*);
  for(*init*;*condition*;*incrementation*){...}

- Foreach ... in
  for(*decl/var* :*array/iterable*){...}

```
public static void main(String[] args) {
  for(String s:args)
    System.out.println(s);
}
```

break/continue label

- **break** [label]:
  permet d'interrompre le bloc
  (en cours ou celui désigné par le label).
- **continue** [label]:
  permet de sauter à l'incrémentation suivante

```
public static boolean find(int val,int v1,int v2) {
  loop: for(int i=1;i<=v1;i++)
    for(int j=1;j<=v2;j++) {
      int mult=v1*v2;
      if (mult==val)
        return true;
      if (mult>val)
        continue loop;
    }
  return false;
}
```

On n'utilise pas de flag !!!

- Malgrès ce que l'on vous à raconté, on utilise pas de drapeau (flag) pour sortir des boucles

```
public static int arghIndexOf(int[] array,int value) {
  int i=0;
  boolean notEnd=true;
  while(i<array.length && notEnd) {
    if (array[i]==value) {
      notEnd=false;
    }
    i++;
  }
  if (notEnd)
    return -1;
  return i;
}
```

- Ce code est faux !!

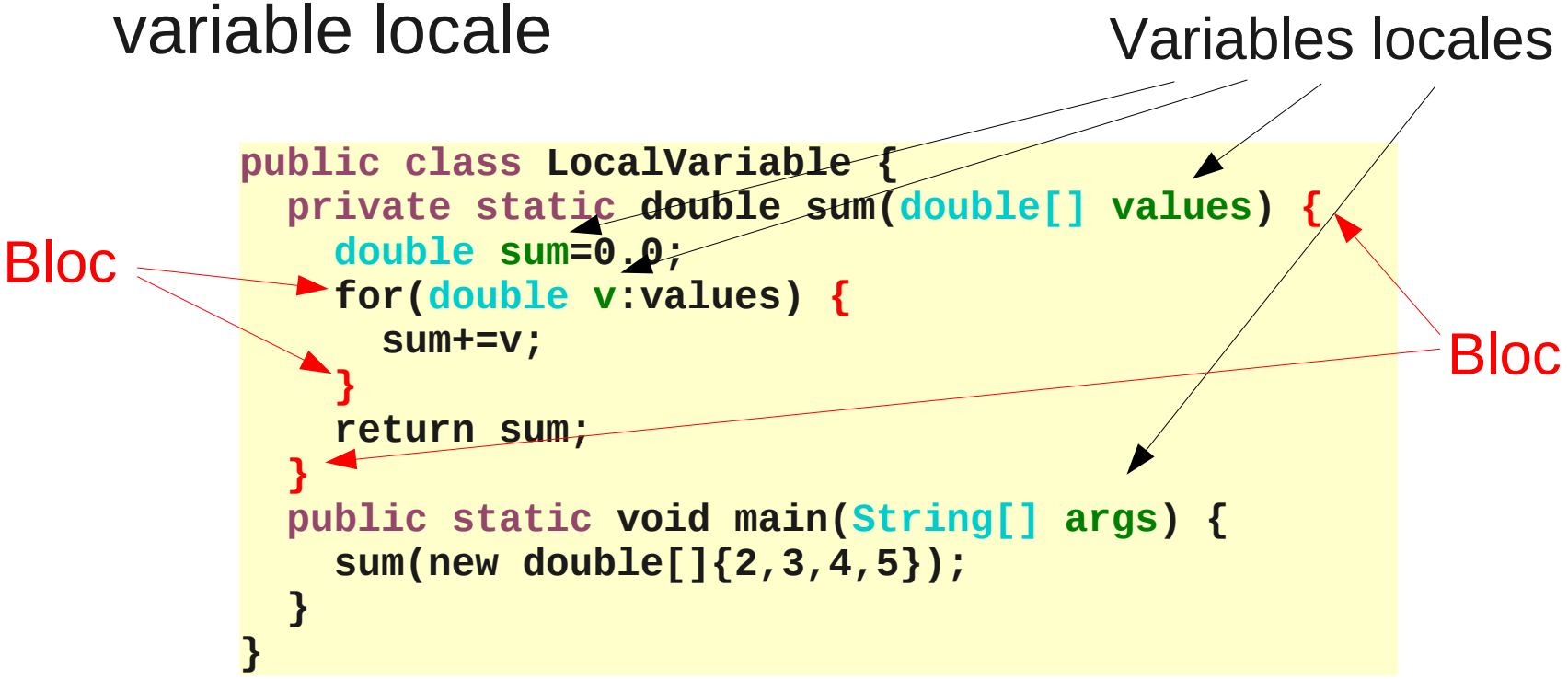
On n'utilise pas de flag !!!

- Utiliser un flag => tours de boucle inutiles, donc des bugs en puissance !!

```
public static int indexOf(int[] array,int value) {
  for(int i=0;i<array.length;i++) {
    if (array[i]==value) {
      return i;
    }
  }
  return -1;
}
```

Variable locale

- Une variable déclarée dans un bloc est une variable locale



- Les paramètres sont aussi considérés comme des variables locales.

Variable locale et portée

- Une variable à pour portée le bloc dans lequel elle est définie

```
public class LocalVariable {
  private static double sum(double[] values) {
    double sum=0.0;
    for(double v:values) {
      sum+=v;
    }           // v n'est plus accessible
    return sum; // values et sum pas accessible
  }
  private static void test(int i) {
    for(int i=0;i<5;i++) // erreur
      doIt(i);
  }
}
```

- Deux variables avec le même nom doivent être dans des blocs disjoints

Variable locale et initialisation

- Toute variable locale doit être initialisée avant son utilisation.

```
public class LocalInit {
  public static void main(String[] args) {
    int value;
    if (args.length==2)
      value=3;
    System.out.println(value);
    // erreur: variable value might not have been initialized
  }
}
```

- Le compilateur effectue la vérification.

Variable locale constante

- Il est possible de définir une variable locale constante en utilisant le mot-clé **final**.

```
public class FinalExample {
  public static void main(String[] args) {
    final int val;
    if (args.length==2)
      val=3;
    else
      val=4;
    // ok pour val ici

    for(final int i=0;i<args.length;i++)
      args[i]="toto";

    // error: cannot assign a value to final variable i
  }
}
```

Variable locale constante (2)

- **final** s'applique sur une variable et non sur le contenu d'un objet

```
public class FinalExample {
  public static void main(final String[] args) {
    final int value;
    if (args.length==2)
      value=3;
    else
      value=4;

    for(int i=0;i<args.length;i++)
      args[i]="toto"; // ok
  }
}
```

- Il est impossible de créer ou d'obtenir un tableau de valeurs constantes en Java !!

Expressions

- Les opérateurs sur les expressions sont les même que ceux du C
  (+,-,*,/, ==, !=, &&, ||, +=, -=, ++, --, etc.)
- Opérateurs en plus
  - >>>, décalage à droite avec bit de signe
  - % sur les flottants ? (float ou double)
- L'ordre d'évaluation des expressions est normé en Java (de gauche à droite)

Modèle d'exécution en mémoire

- Différents types de mémoire d'un programme
  - Texte (donnée read-only) :
    - code du programme, constantes
  - Globale (donnée r/w) :
    - variables globales
  - Pile :
    - une par thread
    - exécute opération courante
  - Tas :
    - objets alloués (par malloc ou new)

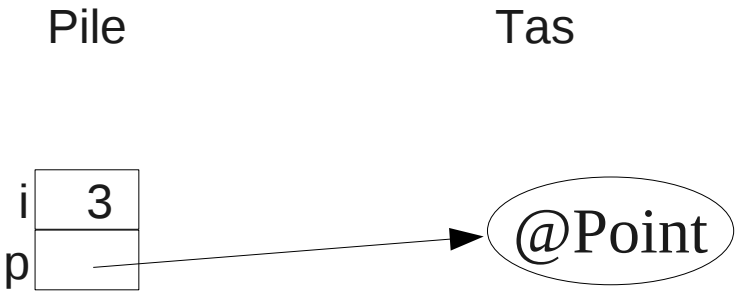
Allocation en Java

- En Java, les variables locales sont allouées sur la **pile** (ou dans les **registres**) et les objets dans le **tas**.
- Le développeur ne contrôle pas la zone d'allocation ni quand sera effectuée la désallocation.
- En C++, il est possible d'allouer des objets sur la pile, pas en Java.

Sur la pile

- La pile, l'ensemble des valeurs courantes d'une exécution
- Sur la pile :
  - Un type primitif est manipulé par sa valeur (sur 32bits ou 64bits)
  - Un type objet est manipulé par sa référence

```
int i;
Point p;

i=3;
p=new Point();
```



Variables locales

La référence **null**

- Java possède une référence spéciale **null**, qui correspond à aucune référence.

```
Point p;
p=null;

System.out.println(p.x);
// lève une exception NullPointerException
```

- **null** est une référence de n'importe quel type.
- **null** instanceof Truc renvoie false

Référence != Pointeur

- Une référence (Java) est une adresse d'un objet en mémoire
- Un pointeur (C) est une référence plus une arithmétique associée

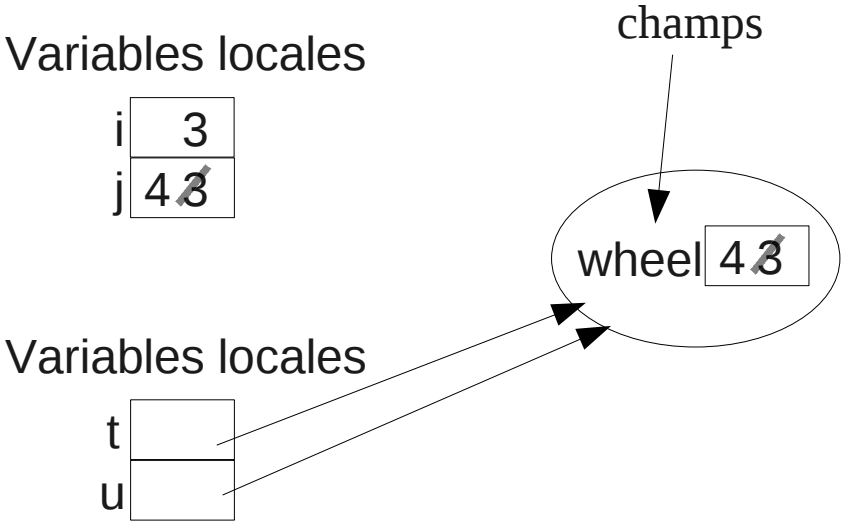
```
Point p;            // Java
p=new Point();
p++;  // illegal

Point *p;           // C
p=new Point();
p++;  // legal
```

- Une référence ne permet pas de parcourir la mémoire

Variable et type

- Simulons l'exécution du code :

```
int i=3;
int j=i;
j=4;
```



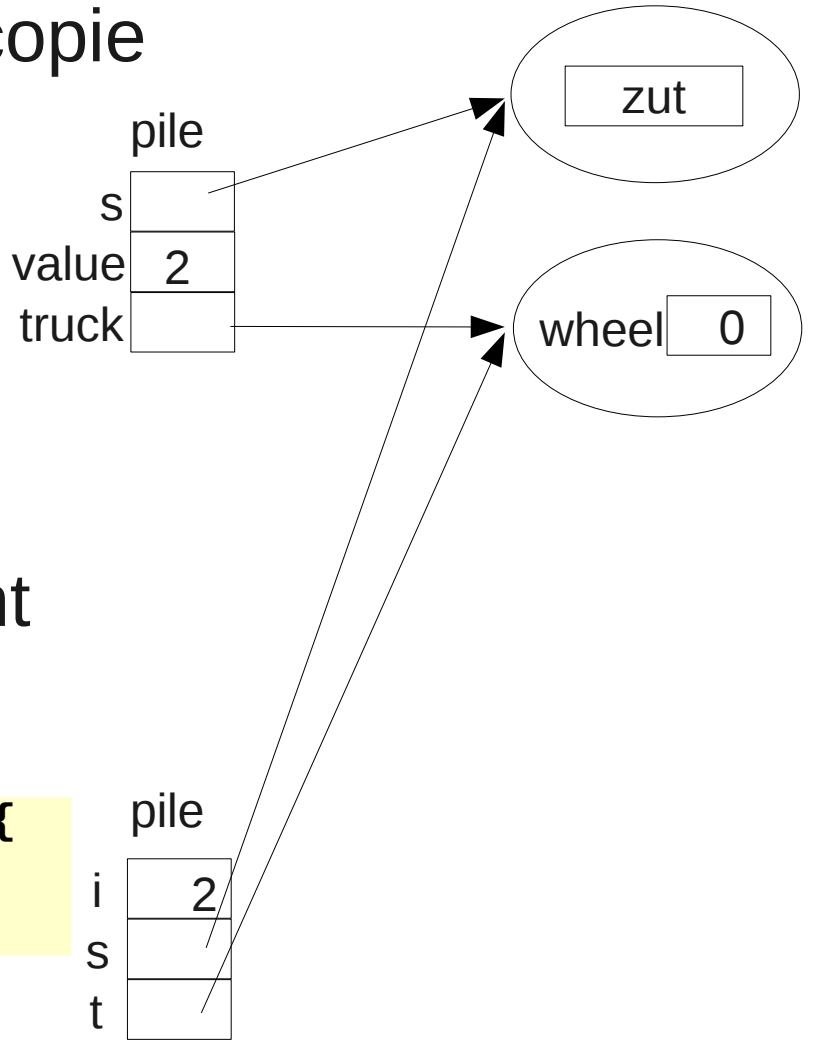

```
Truck t=new Truck();
t.wheel=3;
Truck u=t;
u.wheel=4;
```


- u contient la même référence que t.

Passage de paramètres

- Les paramètres sont passés par copie

```
public static void main(String[] args) {
  String s="zut";
  int value=2;
  Truck truck=new Truck();
  f(value,s,truck);
}
```



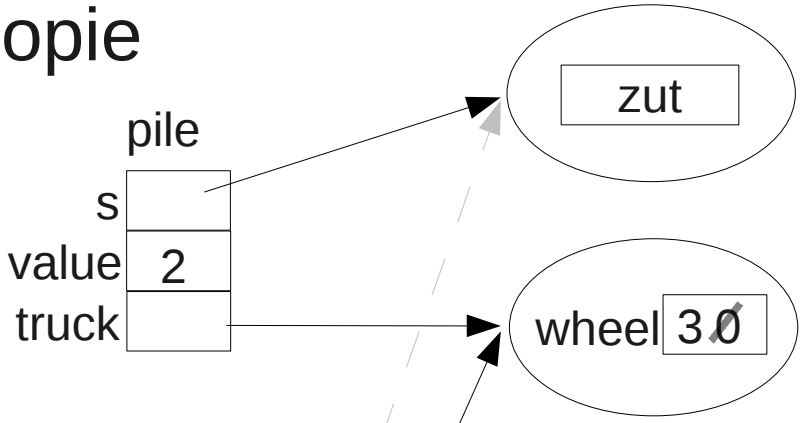
- Pour les objets, les références sont copiées

```
private static void f(int i,String s,Truck t) {
  ...
}
```

Passage de paramètres

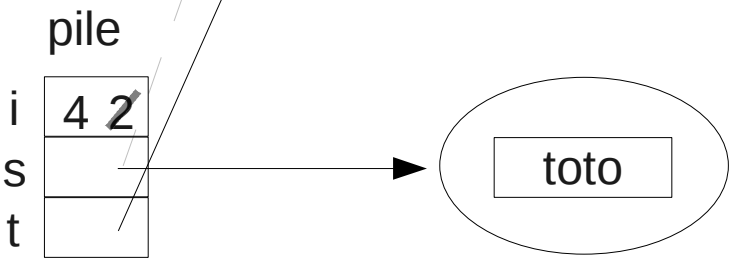
- Les paramètres sont passés par copie

```
public static void main(String[] args) {
  String s="zut";
  int value=2;
  Truck truck=new Truck();
  f(value,s,truck);
}
```



- Une méthode change les variables locales

```
private static void f(int i,String s,Truck t) {
  i=4;
  s="toto";
  t.wheel=3;
}
```



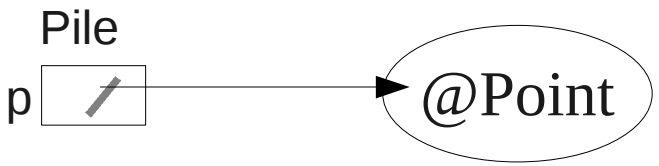
Désallocation en Java

- Les objets qui ne sont plus référencés vont être collectés (plus tard) par le GC pour que leur emplacement mémoire soit réutilisé.
- Façons d'avoir un objet non référencé :
  - Si on assigne une nouvelle référence à une variable (l'ancienne référence “meurt”)
  - Si l'on sort d'un bloc, les variables “meurent”.
- Il faut qu'il n'y ait plus aucune référence sur un objet pour qu'il puisse être réclamé.

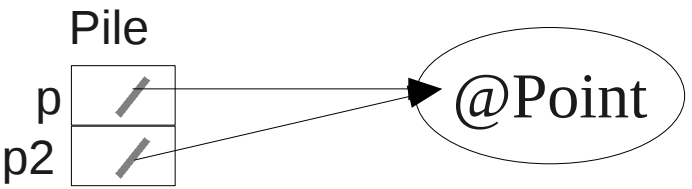
Désallocation en Java (2)

- Un objet sera réclamé par le GC si aucune référence sur lui ne persiste.

```
{
  Point p;
  p=new Point();

  p=null;
}
```




```
{
  Point p;
  p=new Point();

  Point p2;
  p2=p;
  p=null;
}
```



Définition d'une classe en Java

- En Java, “class” permet de définir une classe

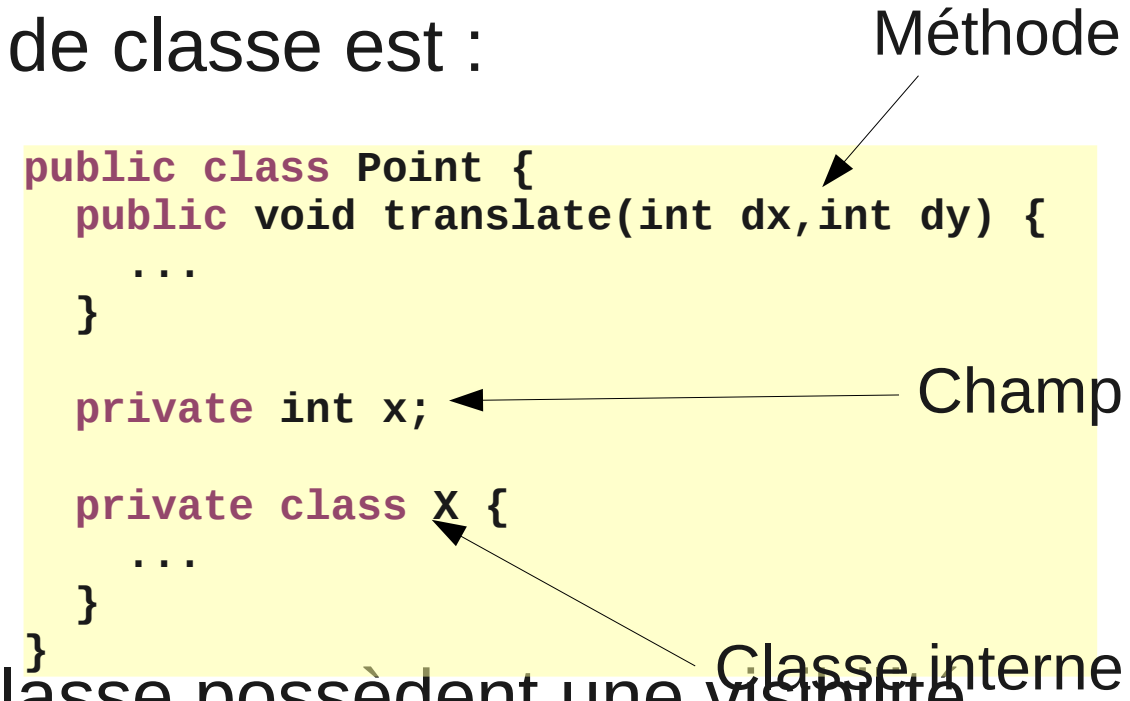
```
class Point {
  void translate(int dx,int dy) {
    x+=dx;
    y+=dy;
  }

  int x;
  int y;
}
```

- La classe Point doit être écrite dans le fichier Point.java

Membres de classe

En java, un membre de classe est :

- Un champ
- Une méthode
- Une classe



- Les membres de classe possèdent une visibilité

Accès aux champs

- L'accès au membre se fait avec « . »

```
public class Foo {
  private int zorg;

  public static void main(String[] args) {
    Foo foo=new Foo();
    foo.zorg=3;
    foo.glub=4;   // glub existe pas
    foo.zorg=5.0; // double -> int, erreur
  }
}
```

- Le compilateur regarde le type de la variable (ici foo) et vérifie que le membre existe bien et qu'il est du bon type

Accès aux méthodes

- L'accès au membre se fait avec « . »

```
public class Foo {
  private int zorg;
  private int zorg(int glub) {
    return glub*2;
  }
  public static void main(String[] args) {
    Foo foo=new Foo();
    foo.zorg=3;              // accès au champ
    int value=foo.zorg(4);  // appel la méthode
  }
}
```

- Un champs et une méthode peuvent avoir le même nom

Instanciation d'une classe

- Créer un objet (une instance) à partir d'une classe (l'instanciation) s'effectue avec “new”

```
public class Point {
  public void translate(int dx,int dy) {
    x+=dx;
    y+=dy;
  }

  private int x;
  private int y;
}
```

```
...
{
  Point p=new Point();
  System.out.printf("%d %d\n",p.x,p.y);
  p.translate(2,3);
  System.out.printf("%d %d\n",p.x,p.y);
}
```

- Les **champs** sont initialisés avec zéro (null, 0, 0.0, false).

Les variables

- Il existe deux types de variables :
  - Les champs
    - sont allouées dans le tas
    - ont la durée de vie de l'objet
    - sont initialisées par défaut
  - Les variables locales
    - sont allouées sur la pile
    - ont la durée de vie de l'appel de méthode
    - ne sont pas initialisées par défaut
      (mais doivent l'être avant le premier accès)

```
public class Foo {
  int zorg; // champ, attribut

  public void m() {
    int glub; // variable locale

    System.out.println(zorg); // 0
    System.out.println(glub);
    // glub pas initialisée
  }
}
```

L'objet courant (**this**)

- Les membres non statique d'une classe possèdent une référence sous-entendu. Vers l'instance courante

- Cet objet est noté "**this**".

```
public class Holder {
   ...
   public void print() {
     System.out.println(this);
   }
   ...
}
```

```
Holder h=new Holder();

System.out.println(h);  // affiche l'adresse de h
h.print();              // affiche l'adresse de h
```

this et les champs

- Autre exemple d'utilisation de **this** :

```
public class Holder {
  public int getValue() {
    return this.value;
    // return value
  }
  public void setValue(int value) {
    this.value=value;
  }
  private int value;
}
```

```
Holder h1=new Holder();
Holder h2=new Holder();

h1.setValue(3);
h2.setValue(5);

System.out.println(h1.getValue());
System.out.println(h2.getValue());
```

- **this** permet de faire référence aux champs en utilisant la notation '**.**'.

Bloc d'initialisation

- Il est possible de définir un bloc de code qui sera exécuté après chaque constructeur.

```
public class Point {
  private double x;
  private double y;
  public Point() {
    x=3;
  }
  public Point(int x) {
    this.x=x;
  }

  {
    x=2;
    y=3;
  }
  public static void main(String[] args) {
    System.out.println(new Point().x);
    System.out.println(new Point(5).x);
  }
}
```

Ordre d'initialisation des champs

- Les champs sont initialisés dans l'ordre de déclaration

```
public class Foo {
  int first=2;
  int second=first;
}
```

```
public class Foo {
  int second=first; // cannot reference a field
                    // before its defined
  int first=2;
}
```

```
public class Foo {
  int second;
  {
    second=first;
  }
  int first=2;
}
```

Ordre d'initialisation

- En Java, il n'y a pas d'ordre imposé entre la déclaration d'un membre et sont utilisation dans une méthode

```
public class Point {
  public void translate(int dx,int dy) {
    x+=dx;
    new X();
  }

  private int x;

  private class X {
    ...
  }
}
```

- Le compilateur effectue 2 passes.

Membre statique

- Une classe a la possibilité de déclarer des membres statiques (mot-clef **static**), qui sont lié à la classe et non à une instance particulière :
  - Champs
  - Méthodes
  - Classes interne statique
- Les membres statiques sont utilisés sans instance de la classe

Contexte statique

- Tout code utilisé dans les membres statiques est utilisé dans un contexte statiques *i.e.* il ne peut faire référence à l'instance courante (**this**).

- Le bloc statique, les interfaces, les enums et les annotations définissent aussi un contexte statique

Champs statiques

- Un champ statique est un champ qui n'est pas propre à un objet mais commun à l'ensemble des objets d'une classe.

```
public class StaticTest {
  private int value;
  private static int staticValue;
  public static void main(String[] args) {
    StaticTest st1=new StaticTest();
    StaticTest st2=new StaticTest();
    System.out.println(++st1.value);                  // 1
    System.out.println(++StaticTest.staticValue);  // 1
    System.out.println(++st2.value);                  // 1
    System.out.println(++StaticTest.staticValue);  // 2
  }
}
```

Accès à un champ statique

- On accède à un champ statique à partir du nom de la classe
- Attention : Le compilateur considère l'accès en utilisant un objet comme légal !!

```
public class StaticTest {
  private static int staticValue;
  public static void main(String[] args) {
    StaticTest st1=new StaticTest();
    System.out.println(++StaticTest.staticValue);
    System.out.println(++st1.staticValue);    // Argh
  }
}
```

Autre exemple

- Chaque objet possède un identifiant (id) unique.

```
public class MyObject {
  private int id=++count;
  private static int count;

  public static void main(String[] args) {
    MyObject o1=new MyObject();
    MyObject o2=new MyObject();
    System.out.printf("%d %d\n",o1.id,o2.id); // 1 2
  }
}
```

Chargement des classes

- En Java, les classes ne sont chargées que si nécessaire

```
public class ClassLoadingExample {
  public static void main(String[] args) {
    if (args.length!=0)
      new AnotherClass();
    System.out.println(args.length);
  }
}
```

- AnotherClass n'est chargée que si **`args.length!=0`**

```
java -verbose:class ClassLoadingExample
[Loaded ClassLoadingExample from file:/C:/java-avancé/]
0
java -verbose:class ClassLoadingExample test
[Loaded ClassLoadingExample from file:/C:/java-avancé/]
[Loaded AnotherClass from file:/C:/java-avancé/]
1
```

Initialisateur de classe

- Le bloc statique sert à déclarer un code qui sera exécuté une fois lors de l'initialisation de la classe.

```
public class HelloStatic {
  public static void main(String[] args) {
    System.out.println(hello);
  }
  private static final String hello;
  static {
    char[] array=new char[]
{'h','e','l','l','o'};
    hello=new String(array);
  }
}
```

- Bloc statique <=> constructeur de classe

Allocation paresseuse

- Il est possible de profiter du chargement “à la volée” des classes.

```
class HolderCache {
  static Holder[] holders=new Holder[255];
  static {
    for(int i=0;i<holders.length;i++)
      holders[i]=new Holder(i);
  }
}
public class LazyLoading {
  public static Holder getHolder(int value) {
    if (value>=0 && value<=255)
      return HolderCache.holders[value];
    return new Holder(value);
  }
}
```

Les constantes

- Définir une constante en C, on utilise **#define**

```
public class ConstExample {
  public static String getLine(Scanner scanner) {
    return scanner.findWithinHorizon("[a-z]+", MAX_SIZE);
  }
  private static final int MAX_SIZE=4096;
}
```

- En Java, on utilise une variable **static** et **final** (donc une variable typée).

Entrée/sortie

- System.in représente entrée standard
- System.out la sortie standard
- System.err la sortie d'erreur standard

- Ce sont des champs **static final** de la classe java.lang.System

```
PrintStream out=System.out;
out.printf(“%s “,“hello”);
out.println(“world”);
```

Constante et **#ifdef**

- Évaluation des constantes est effectuée au chargement de la classe.

```
public class IfDef {
  public static void test() {
    for(int i=0;i<1000000;i++) {
      if (DEBUG)
        System.out.println(i);
    }
  }
  private static final boolean DEBUG=false;
  public static void main(String[] args) {
    test();
  }
}
```

Constante comme #ifdef

- Vitesse d'exécution (nano-benchmark) :

```
public class IfDef {
  public void test() {
    for(int i=0;i<1000000;i++) {
      if (DEBUG)
        System.out.println(i);
    }
  }
  private static final boolean DEBUG=false;
  public static void main(String[] args) {
    IfDef ifdef=new IfDef();
    long time=System.nanoTime();
    ifdef.test();
    long time2=System.nanoTime();
    System.out.println(time2-time);
  }
}
```

Temps si **if** commenté : 2,2ms

Temps si **if** présent :

		final
	3,4 ms	2,2 ms
static	3,3 ms	2,2 ms

Méthode Statique

Une méthode statique est une méthode qui peut-être appelée sans nécessité d'objet
(comme une fonction en C !)

```
public class Sum {
  private static void sum(int[] values) {
    int sum=0;
    for(int v:values)
      sum+=v;
    return sum;
  }
  public static void main(String[] args) {
    Sum.sum(new int[]{2,3,4,5});
    // ou sum(new int[]{2,3,4,5});
  }
}
```

Méthode statique (2)

Une méthode statique n'a accès qu'aux champs statiques d'une classe.

```
public class Point {
  private static int test() {
    int v=value; // legal
    return x+y;  // illegal, quel objet Point utilisé ?
  }
  private int x,y;
  private static double value;
}
```

Une méthode statique ne possède pas d'objet courant (pas de **this**)

Singleton

- Les méthodes statiques sont souvent utilisés pour assurer l'unicité d'un objet

```
public class Platform {
  public static Platform getDefaultPlatform() {
    return platform;
  }
  private static Platform platform=
    new Platform();
}
```

- Base du design-pattern singleton

```
Platform platform = Platform.getDefaultPlatform();
```

Paquetage

- Un paquetage (*package*) est un regroupement de classes ayant trait à un même concept
- Un paquetage :
  - sert à eviter les classes de même nom
  - doit être déclaré au début de chaque classe
  - correspond à un emplacement sur le disque ou dans un jar (classpath)

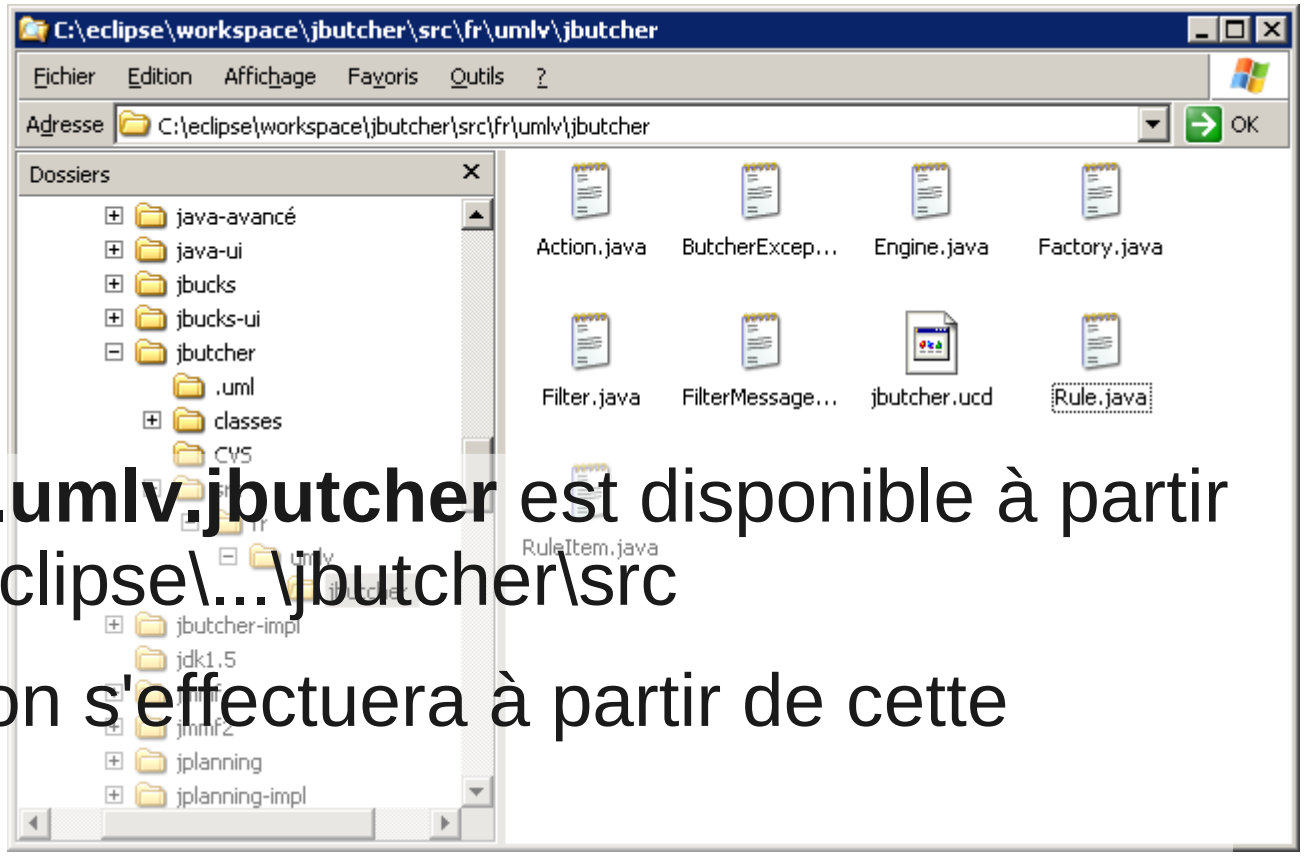
Déclaration d'un paquetage

- Déclaration de classe avec paquetage :

```
package fr.umlv.jbutcher;

public classe Rule {
  ...
}
```

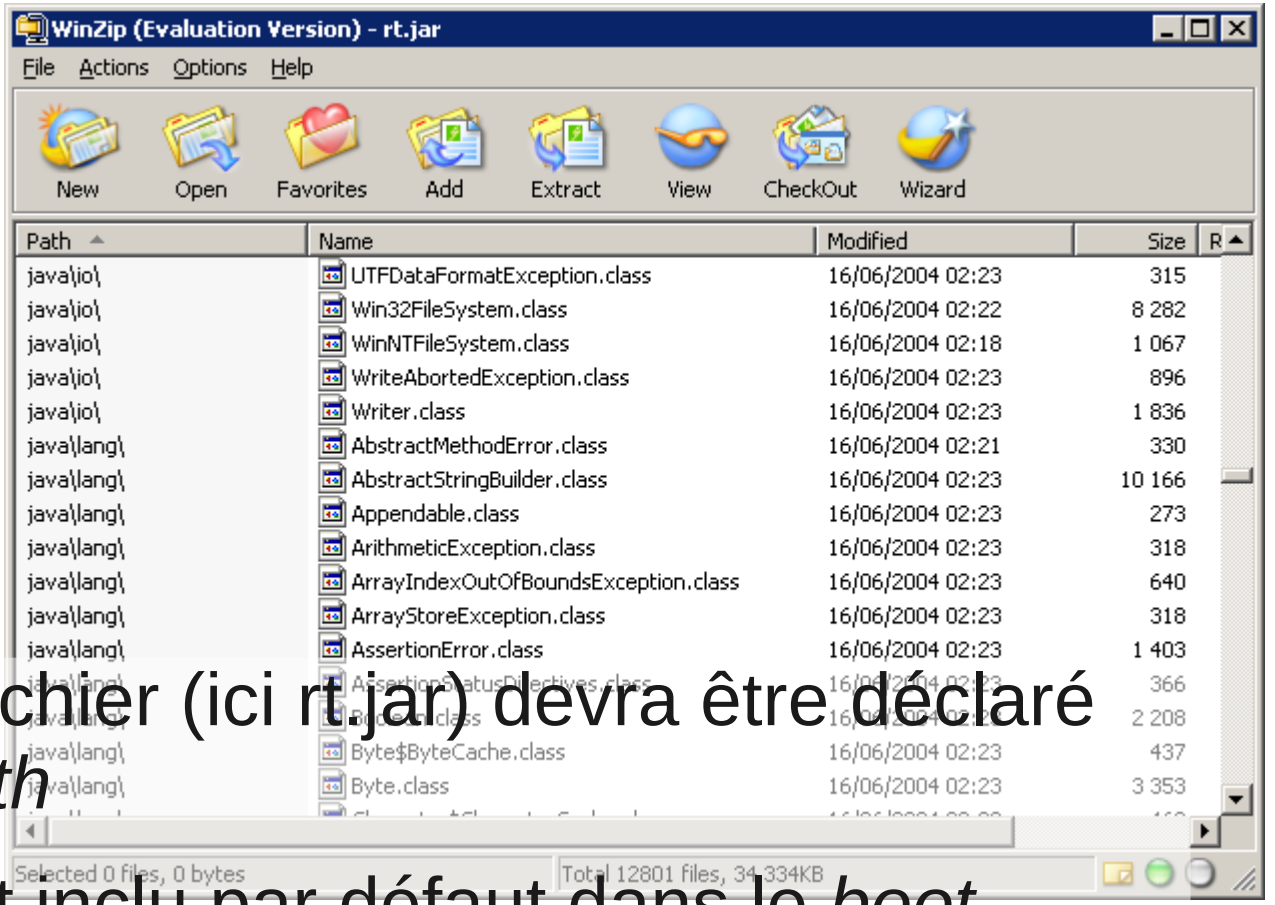
- Le nom de la classe est **fr.umlv.jbutcher.Rule**
- Règle de nommage :
  fr.masociété.monprojet.monmodule
  ex: com.google.adserver.auth

Organisation sur le disque



- Le paquetage **fr.umlv.jbutcher** est disponible à partir de l'adresse c:\eclipse\...\jbutcher\src
- Soit la compilation s'effectuera à partir de cette adresse
- Soit la variable CLASSPATH pointera sur cette adresse

Organisation dans un jar



- Pour un jar, le fichier (ici rt.jar) devra être déclaré dans le *classpath*
- En fait, **rt.jar** est inclu par défaut dans le *boot classpath*

La directive **import**

- La directive **import** permet d'éviter de nommer une classe avec son paquetage

```
public class MyButcher {
  public static void main(String[] args) {
    fr.umlv.jbutcher.Rule rule=
      new fr.umlv.jbutcher.Rule();
  }
}
```

```
import fr.umlv.jbutcher.Rule;

public class MyButcher {
  public static void main(String[] args) {
    Rule rule=new Rule();
  }
}
```

- Le compilateur comprend que **Rule** à pour vrai nom **fr.umlv.jbutcher.Rule**
- Le bytecode généré est donc identique

Import *

- Indique au compilateur que s'il ne trouve pas une classe, il peut regarder dans les paquetages désignés

```
import java.util.*;
import fr.umlv.jbutcher.*;

public class MyButcher {
  public static void main(String[] args) {
    ArrayList list=new ArrayList();
    Rule rule=new Rule();
  }
}
```

- Ici, **ArrayList** est associé à **java.util.ArrayList** et **Rule** à **fr.umlv.jbutcher.Rule**

Import * et ambiguité

- Si deux paquetages possèdent une classe de même nom et que les deux sont importés en utilisant *, il y a ambiguïté

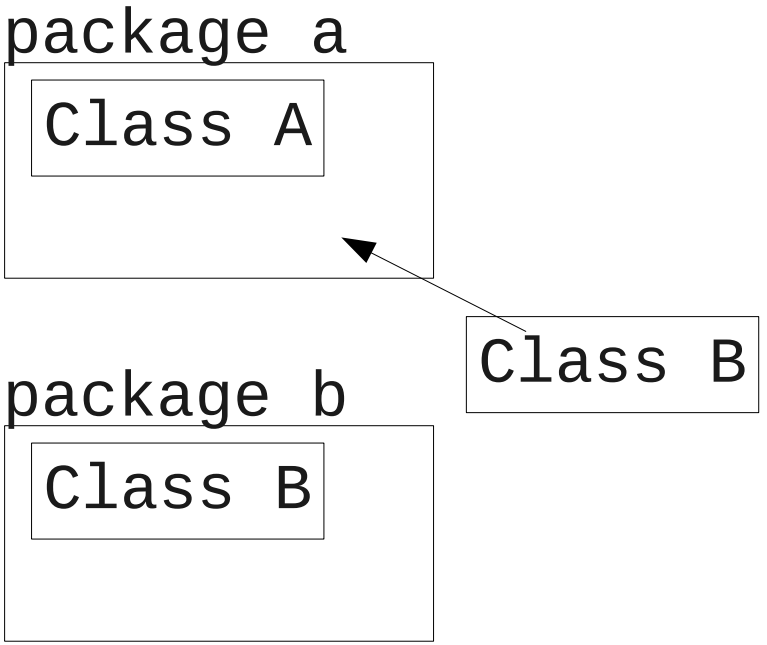
```
import java.util.*;
import java.awt.*;

public class ImportClash {
  public static void main(String[] args) {
    List list=... // oups
  }
}
```

```
import java.util.*;
import java.awt.*;
import java.util.List;

public class ImportClash {
  public static void main(String[] args) {
    List list=... // ok
  }
}
```

Import * et maintenance

- **import *** pose un problème de maintenance si des classes peuvent être ajoutées dans les paquetages utilisés

```
import a.*;
import b.*;

public class ImportClash {
  public static void main(String[] args) {
    A a=new A();
    B b=new B();
  }
}
```



- Règle de programmation : éviter d'utiliser des **import ***

Import statique

- Permet d'accéder aux membres statiques d'une classe dans une autre sans utiliser la notation '.'

```
import java.util.Scanner;
import static java.lang.Math.*;

public class StaticImport {
  public static void main(String[] args) {
    Scanner in=new Scanner(System.in);
    System.out.println("donner un nombre :");
    double value=sin(in.nextDouble());
    System.out.printf("son sinus est %f\n",value);
  }
}
```

- Notation : **import static chemin.classe.*;**

Import Statique et scope

- Lors de la résolution des membres, les membres (même hérités) sont prioritaires sur le scope

```
import static java.util.Arrays.*;

public class WeirdStaticImport {
  public static void main(String[] args) {
    java.util.Arrays.toString(args); // ok
    toString(args); // toString() in java.lang.Object
                    // cannot be applied to (java.lang.String[])
  }
}
```

- Règle de programmation :
  utiliser l'import statique avec parcimonie

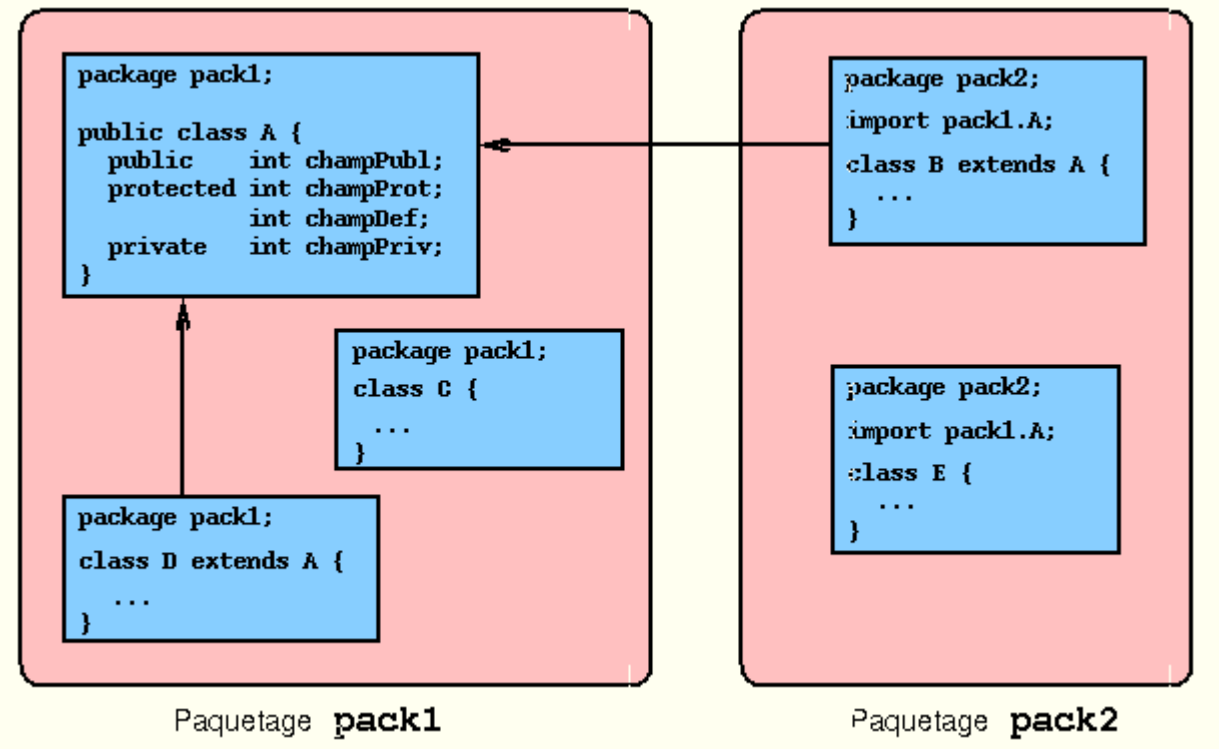
Principe de protection

- Permet d'empêcher l'accès à des membres en fonction de la classe qui déclare le membre et de la classe qui y accède
- Très important :
  - Permet l'encapsulation et la protection des données
  - Permet une bonne maintenance en exposant le moins de choses possibles
- L'accès est vérifié par la VM

Modificateur de visibilité

- Il existe 4 modificateurs de visibilité :
  - un membre **private**, n'est visible qu'à l'intérieur de la classe.
  - un membre **sans modificateur** est visible par toute les classes du même package.
  - un membre **protected** est visible par les classes héritées et celles du même package.
  - un membre **public** est visible par tout le monde.
- Il existe un ordre de visibilité
  private< <protected<public

exemple



Dans :	A	B	C	D	E
champPubl	oui	oui	oui	oui	oui
champProt	oui	oui	oui	oui	-
champDef	oui	-	oui	oui	-
champPriv	oui	-	-	-	-

Visibilité et maintenance

- Quelques règles :
  - Un champ n'est jamais **protected** ou public (sauf les constantes)
  - La visibilité de paquetage est utilisée si une classe partage des détails d'implantation avec une autre (exemple: des classes internes)
  - Une méthode n'est **public** que si nécessaire
  - Une méthode **protected** permet d'implanter un service commun pour les sous classes, si celui-ci ne peut être écrit hors de la classe avec une visibilité de paquetage.