Modèle d'exécution

Plan

- Types primitifs
- Methode, variable locale, initialisation
- Champs statique & constantes
- Package

Les Types et Java

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 (non signé!): 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)

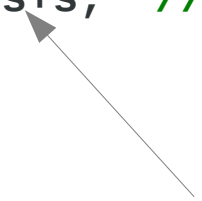
- Unix (dépend ISO-Latin1, UTF8, UTF16)
- Windows (Windows Latin1)
- Mac-OS (MacRoman, ...)

Les valeurs entières signés

Les valeurs sont **signées** :
byte va de -128 à 127
short de -32 768 à 32 767

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
}
...
```

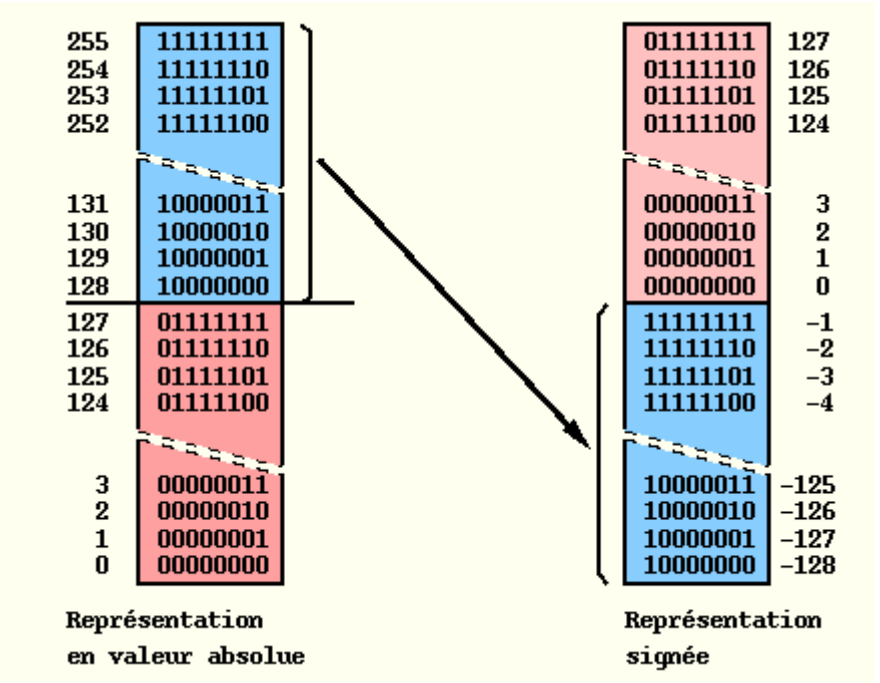


Le résultat est un int (promotion entière)

Les valeurs entières signés (2)

Valeur signée et représentation en complément à 2.

Le cast préserve la valeur pas les bits




```
...
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

Constant entière

3 dans a = 3 n'est pas un int en Java,
la valeur dépend tu type de a

```
byte b = 3;  // ok
short s = 3; // ok
int s = 3; // ok
long l = 3; // ok
```

Ce mécanisme n'existe pas pour les valeurs flottantes

```
float f = 3.0;  // compile pas
```

Division entière

Il n'est pas possible de diviser un int ou un long (promotion entière) par 0

L'exception DivideByZeroException est levée

Il n'y a pas ce genre de problème avec les flottants !

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);
}
```

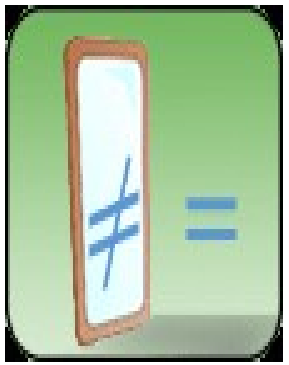
```
Infinity
NaN
-Infinity
NaN
```

- 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 sur 80 bits
- 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
    // utiliser plutot '<'
```

Strictfp

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

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

stricfp induit un cout lors de l'exécution

Attention aux arrondies !

Les fottants sont une approximation des nombres 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** réprésentant 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

Méthode & code

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**(string/enum)
- **for** spécial dit “foreach” for(Type item : array)
- **break** ou **continue** avec label.
- **try** {} **catch**() {} **finally** {}
- try with resources: **try**(A a = ...) { ... }

Structure de test

Test Conditionnel booléen :

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

Test Multiple numérique ou string ou enum:

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

}
```

```
switch(args[0]) {
  case "-a":
    lsAll();
    break;
  default:
    lsSimple();
}
```

```
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 !! (et illisible)

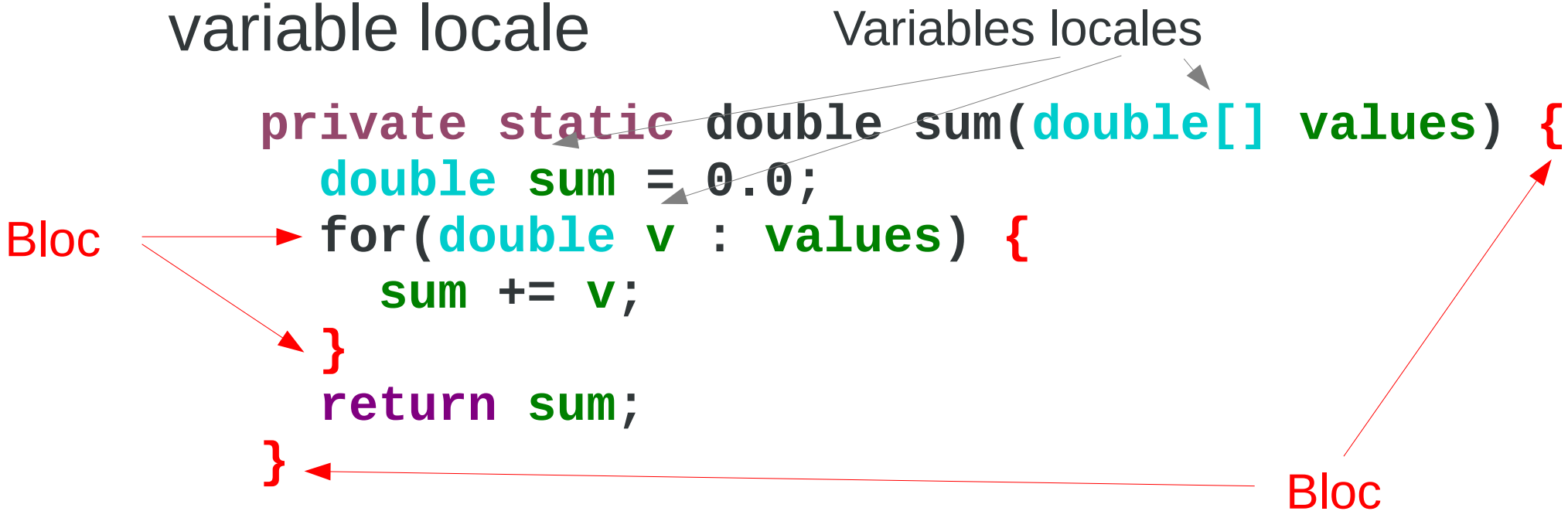
On n'utilise pas de flag !!!

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

```
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 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 non modifiée

Il est possible de dire qu'une variable locale est intialisé une seule dois avec le mot-clé **final**.

```
public static void main(String[] args) {
   final int value;
   if (args.length == 2) {
     value = 3;
   } else {
     value = 4;
   } // ok pour value ici
   for(final int i = 0; i < args.length; i++) {
     args[i] = "toto";
   }
   // error: cannot assign a value to final variable i
}
```

Variable locale final (2)

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

```
public static void main(final String[] args) {
  for(int i = 0; i < args.length; i++) {
    args[i] = "toto"; // ok
  }
}
```

final sur les variables locales est une informations pour le compilateur pas la VM

Variable locale final et légende urbaine

Mettre final sur une variable locale ne change rien pour la vitesse d'exécution

final est une information pour le compilateur, cette information n'est pas stocké dans le .class

final sur les variables locales est **peu utilisé** dans la vrai vie
(contrairement à final sur les champs)

Variable locale constante (2)

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

```
public static void main(final String[] args) {
  for(int i = 0; i < args.length; i++) {
    args[i] = "toto"; // ok
  }
}
```

final sur les variables locales est une informations pour le compilateur pas la VM

final sur les variables locales est **peu utilisé** (contrairement à final sur les champs)

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)

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/C++, il est possible d'allouer des objets sur la pile, pas en Java

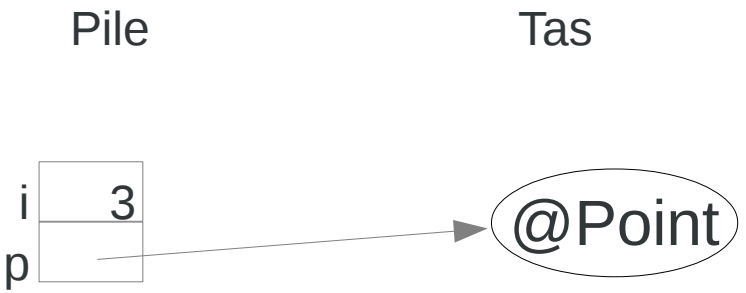
Sur la pile

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

Référence != Pointeur

Une référence (Java) est une adresse d'un objet en mémoire

Un pointeur (C) est une référence plus une arithmétique associée

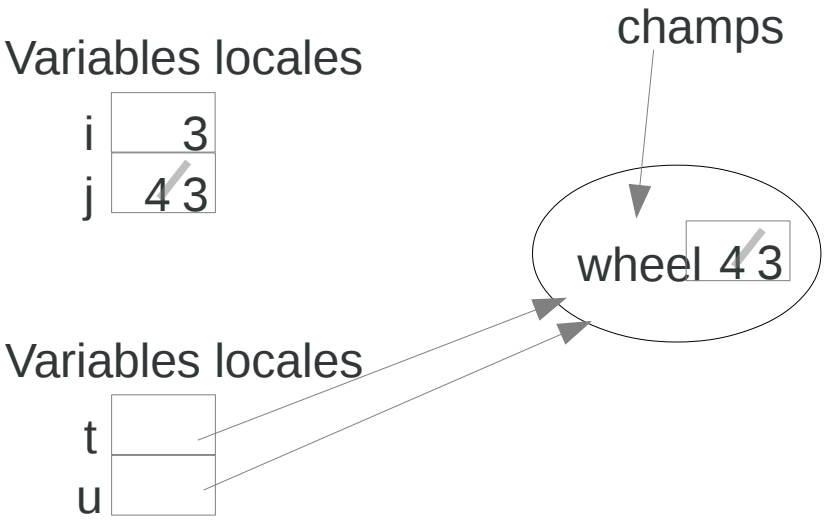
```
Point p;          // Java
P = new Point();
p++;  // illegal

Point *p;         // C
p = (Point*)malloc(sizeof(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;
```



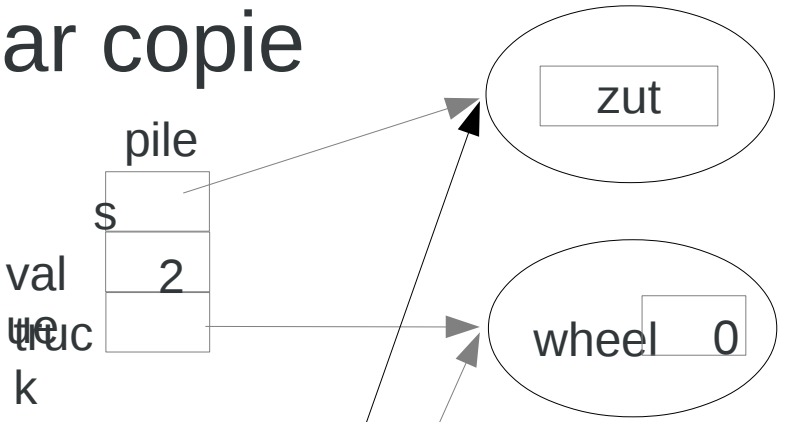

```
Truck t = new Truck();
t.wheel = 3;
Truck u = t;
u.wheel = 4;
```


u contient la même référence que t.

Passage de paramètres

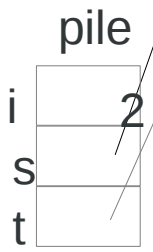
Les paramètres sont passés par copie

```
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

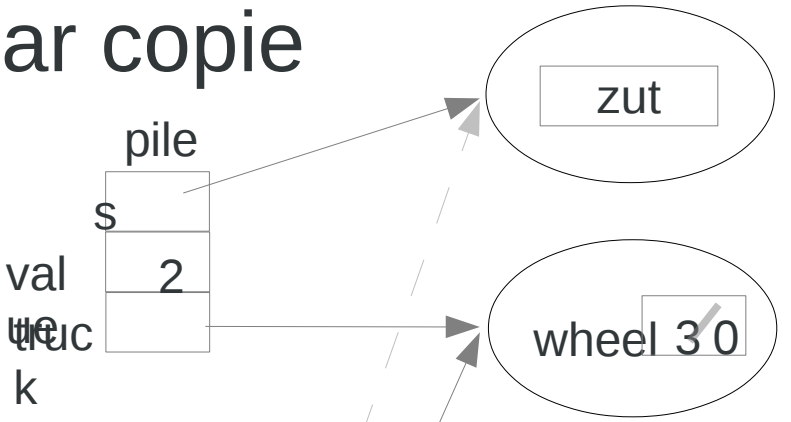
```
private static void f(int i,String s,Truck t) {
  ...
}
```



Passage de paramètres

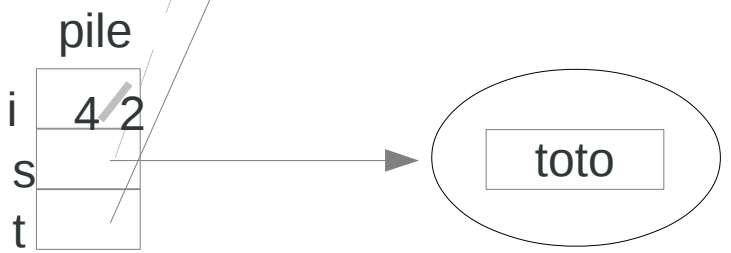
Les paramètres sont passés par copie

```
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;
}
```



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 Foo renvoie false

Rêgles d'utilisation de null

Le problème de null est que si une méthode renvoie null, comme les developpeurs ne lise pas la documentation, le risque de NPE est grand

```
car.getOwner().getName()  // NullPointerException
```

Il faut donc des rêgles de bonne pratique d'utilisation de null.

si un code a des if (var == null) partout, c'est qu'il y a un gros problème !

Sémantique de null

Malheureusement, null est pratique pour dire qu'il n'y a pas d'élément

- Une voiture n'a pas de caravane attachée,
- Un arbre rouge/noir (TreeSet) n'a pas de fonction de comparaison
- etc.

C'est la seule sémantique de null acceptable !

Rêgles d'utilisation

Par défaut :

- Une variable local ne doit pas être null
- Un champs ne peut pas être null
  - On vérifie à la création (et dans les setters) avec java.util.Objects.requireNonNull
- Une méthode ne doit pas renvoyer null

si on passe outre une des rêgles ci-dessus

on doit avoir un commentaire de documentation

et on doit avoir un commentaire dans le code

null et les variables locales

null **ne doit pas** être utilisé dans les variables locales pour indiquer qu'une variable n'est pas initialisée

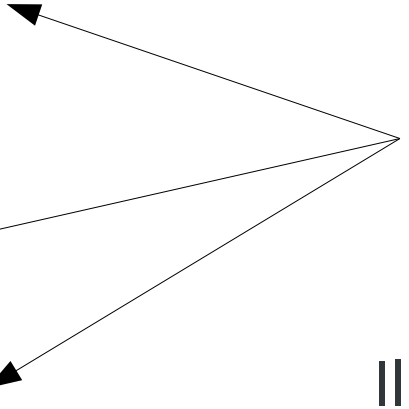
```
Car car = null;
if (emergency) {
   car = new Car(owner);
} else {
   car = new PoliceCar();
}
```

Ahhhhh,
pas nécessaire

On peut déclarer
une variable sans l'initialiser
Car car;

Champs à null

```
public class Car {
  private final String owner;
  private Trailer trailer;

  public Car(String owner) {
    this.owner = owner;
  }
  public String getOwner() {
    return owner;
  }
  public void setTrailer(Trailer trailer) {
    this.trailer = trailer;
  }
  public Trailer getTrailer() {
    return trailer;
  }
}
```



Il est plus simple de **ne pas permettre d'insérer null** plutôt que de tester si la valeur est null à chaque fois que l'on appel getOwner()

Champs à null

```
public class Car {
  private final String owner;
  private Trailer trailer;  // may be null

  public Car(String owner) {
    this.owner = Objects.requireNonNull(owner);
  }
  public String getOwner() {
    return owner;
  }
  /** Set the trailer or null if there is no trailer.
   */
  public void setTrailer(Trailer trailer) {
    this.trailer = trailer;
  }
  /** Get the trailer of the current car or null.
   */
  public Trailer getTrailer() {
    return trailer;
  }
}
```

commentaire dans le code

commentaire de documentation

Collection et null

Dans le cas où on veut dire que l'on a pas d'élement dans un structure de donnée, on utilise pas null mais une collection vide

- ArrayList et HashMap sont optimisées en mémoire pour ce cas
- Il existe Collections.emptyList(), Collections.emptySet() et Collection.emptyMap() pour les autres implantations

On ne **renvoie jamais null** lorsque l'on attend une collection

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 est “perdu”)
- Si l'on sort d'un bloc, les variables locales “meurent”.

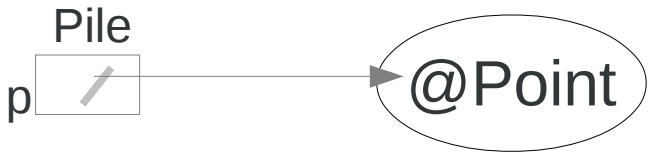
Il faut qu'il n'y ait plus aucune référence sur un objet pour qu'il puisse être réclamé.

Désallocation en Java (2)

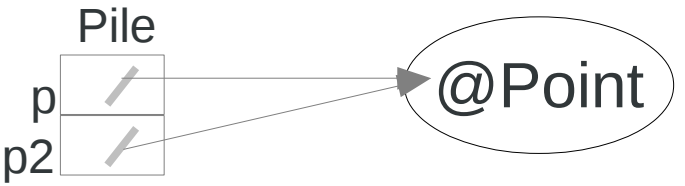
Un objet peut être réclamé par le GC à partir du moment où plus aucune référence sur celui-ci existe persiste.

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

On sort du bloc, p existe plus




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

  Point p2;
  p2 = p;
}
```



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
  }
}
```

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

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

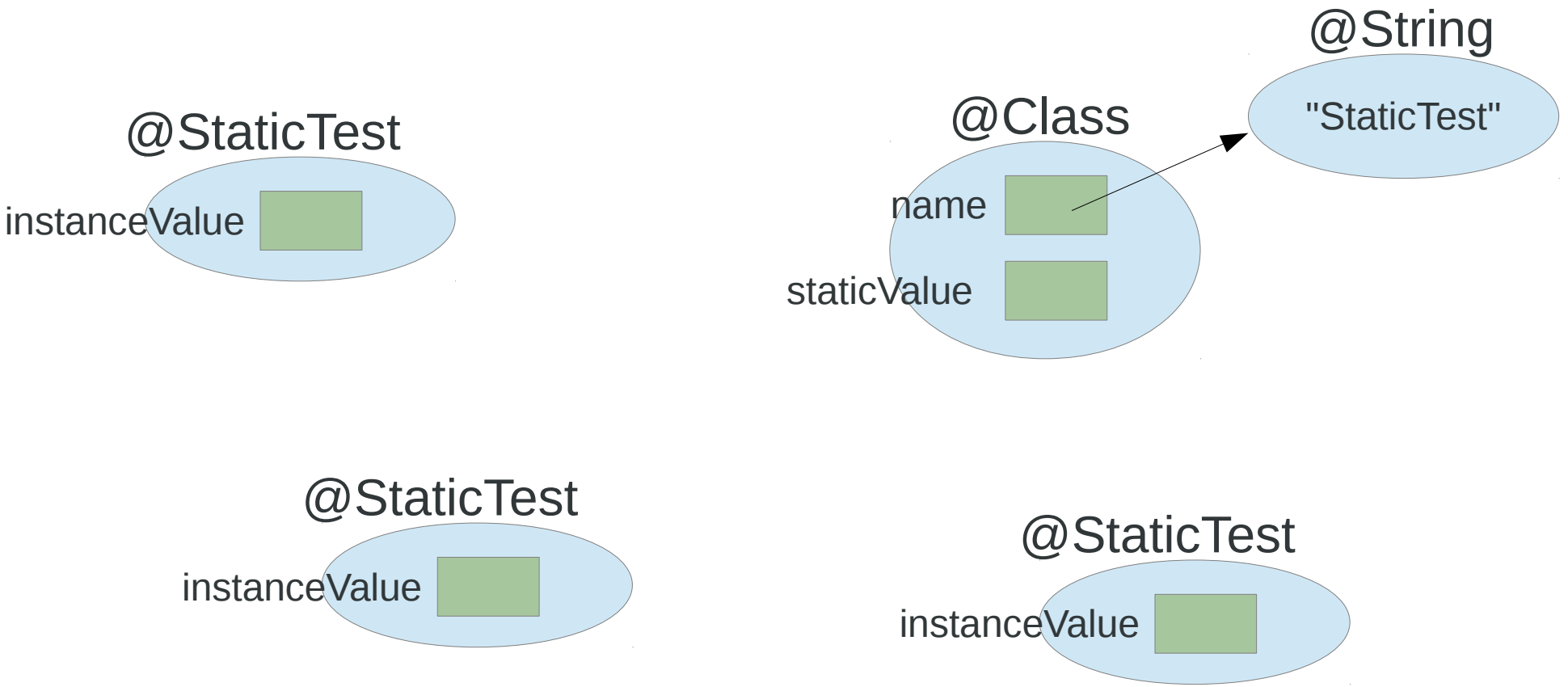
Un champ statique est un champ qui n'est pas propre à un objet mais commun à l'ensemble des objets d'une classe

```
public class StaticTest {
  private int instanceValue;
  private static int staticValue;

  public static void main(String[] args) {
    System.out.println(++StaticTest.staticValue);  // 1
    System.out.println(++StaticTest.staticValue);  // 2
  }
}
```

La valeur d'un champs statique de type objet ne sera réclamée que si la classe des déchargées (jamais par défaut)

En mémoire ...

Un champs statique est stocké au niveau de la classe et pas au niveau d'une instance



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) {
    System.out.println(++StaticTest.staticValue);

    StaticTest st1 = new StaticTest();
    System.out.println(++st1.staticValue);    // Aaaargh !
  }
}
```

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
  }
}
```

Problèmes des champs statiques

Un champs statique est une valeur globale

- impossible à changer lors de tests

La valeur d'un champs statique **de type objet** ne sera réclamée que si la classe des déchargées (jamais par défaut)

- => memory leak !

On évite d'utiliser des variables statiques !!!

Les constantes

Définir une constante en C, on utilise **#define, e**n Java, on utilise une variable **static et final**

```
public class ConstExample {
  private static final int MAX_SIZE=4096;

  public static String getLine(Scanner scanner) {
    return scanner.findWithinHorizon("[a-z]+", MAX_SIZE);
  }
}
```

Constante et **#ifdef**

Les static final de type primitif ou String sont optimisé par le compilateur

```
public class IfDef {
  private static final boolean DEBUG = false;

  public static void test() {
    for(int i = 0; i < 1_000_000;i++) {
      if (DEBUG) {
        System.out.println(i);
      }
    }
  }
}
```

Ce code n'est pas généré !

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”);
```

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

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
```

Méthode

Les méthodes en Java peuvent être :

- à nombre constant
- à nombre variables d'arguments
- surchargées
  (plusieurs méthodes avec le même nom)

Il existe de plus un type de méthode particulier qui permet d'initialiser un objet appelé constructeur.

Différents types de méthodes

Classification des méthodes d'un objet

- Constructeur
  - Initialise l'objet
- Méthode statique (factory)
- Accesseur
  - Getter (getX)
    - Export les données (souvent partiellement) vers l'extérieur
  - Setter (setX)
    - Import les données (en les vérifiants) de l'extérieur
- Méthode privée (qui peuvent être statique)
- Méthode publique

Constructeur par défaut

Si aucun constructeur n'est défini, le compilateur rajoute un constructeur **public** sans paramètre

```
public class Point {
  private double x;
  private double y;

  public static void main(String[] args) {
    Point p = new Point(); // ok
  }
}
```

Contrairement au C++, Il n'est pas nécessaire de mettre obligatoirement un constructeur par défaut.

Appel inter-constructeur

Appel à un autre constructeur se fait avec la notation **this(...)**

```
public class Counter {
  public Counter(int initialValue) {
    this.counter = initialValue;
  }
  public Counter() {
    this(0);
  }
  public int increment() {
    return counter++;
  }
  private int counter;
}
```

```
public class AnotherClass {
  public static void main(String[] args) {
    Counter c1 = new Counter();
    System.out.println(c1.increment());
    Counter c2 = new Counter(12);
    System.out.println(c2.increment());
  }
}
```

this(...) doit être la première instruction

Constructeur vs Méthode

La grosse différence entre un constructeur et une méthode, c'est que l'**on n'hérite pas des constructeurs**

Le constructeur de la classe hérité a pour mission de :

- demander l'initialisation de la classe mêre au travers de son constructeur
- d'initialiser ses propres champs

Publication unsafe de this

Un constructeur ne doit pas appeler un méthode qui prend this en paramètre car l'objet courant n'est pas fini d'être initialisé

Pour la même raison, on ne doit pas appeler de méthodes que l'on peut redéfinir (par exemple un setter) dans un constructeur

Varargs

Il est possible de définir des méthodes à nombre variables d'arguments en utilisant la notation ...

```
public class PrintStream {
  public void printf(String text, Object... args) {
    ...
  }
}
```

Reçoit les arguments dans un tableau

```
  public static void main(String[] args) {
    PrintStream out=System.out;
    out.printf("%d\n",2);
  }
```

Varargs (2)

La notation ... doit être utilisée que par le dernier argument (comme en C)

```
public static int min(int... array) {
  if (array.length == 0)
    throw new IllegalArgumentException("array is empty");
  int min = Integer.MIN_VALUE;
  for(int i: array) {
    if (i > min) {
      min = i;
    }
  }
  return min;
}
public static void main(String[] args) {
  min(2,3); // tableau contenant deux valeurs
  min(2);   // tableau contenant une valeur
  min();    // tableau vide
  min(new int[]{2}); // utilise le tableau passé
}
```

Varargs et null

Il y a une ambiguité dans le cas d'un Object... si l'on passe **null**

```
public class PrintStream {
  public void printf(String text, Object... args) {
    ...
  }
}
```

```
public static void main(String[] args) {
  PrintStream out=System.out;
  out.printf("",null); // non-varargs call of varargs method with
                       // inexact argument type for last parameter
}
```

Le compilateur émet un warning et envoie **null** comme argument

```
 out.printf("",(Object[])null); // enlève l'ambiguité
```

Varargs & compatibilité

Les varargs sont considérés comme des tableaux par la VM

Il n'est donc pas possible d'effectuer une surcharge entre tableau et varargs :

```
public class VarargsOverloading {
  private static int min(int... array) { }
  private static int min(int[] array) {
  } // min(int...) is already defined in VarargsOverloading
}
```

Surcharge de méthodes

Nommée aussi surdéfinition (*overloading*), consiste à fournir plusieurs définitions pour une méthode

```
public class PrintStream {
  public void println(String text) {
    ...
  }
  public void println(double value) {
    ...
  }
}
```

Le compilateur cherche la meilleure méthode en fonction du type des arguments

```
public static void main(String[] args) {
  PrintStream out=System.out;
  out.println("toto");
  out.println(3.0);
}
```

Surcharge et Typage

Si aucune méthode ne possède les types exactes, le compilateur prend une méthode approchée en fonction du typage.

```
public class PrintStream {
  public void println(String text) {
    ...
  }
  public void println(double value) {
    ...
  }
}
```

```
  public static void main(String[] args) {
    PrintStream out=System.out;
    out.println(3);   // int -> double
    out.println(out); // erreur
  }
```

Limitation sur le type de retour

Le type de retour n'est pas considéré lors d'un appel, il est donc impossible de différencier des méthodes uniquement en fonction de leur type de retour.

```
public class BadOverloading {
  public int f() {
    ...
  }
  public double f() {
    ...
  }
  // f() is already defined in BadOverloading

}
```

```
  public static void main(String[] args) {
    BadOverloading bo = new BadOverloading();
    bo.f();
  }
```

Quand doit-on surcharger ?

Règle empirique : on effectue une surcharge entre deux méthodes si celles-ci ont la même sémantique.

```
public class Math {
  public float sqrt(float value) {
  }
  public double sqrt(double value) {
  } // ok
}
```

```
public class List {
  public void remove(Object value) {
  }
  public void remove(int index) {
  } // c'est Mal, pas la même sémantique
}
```

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

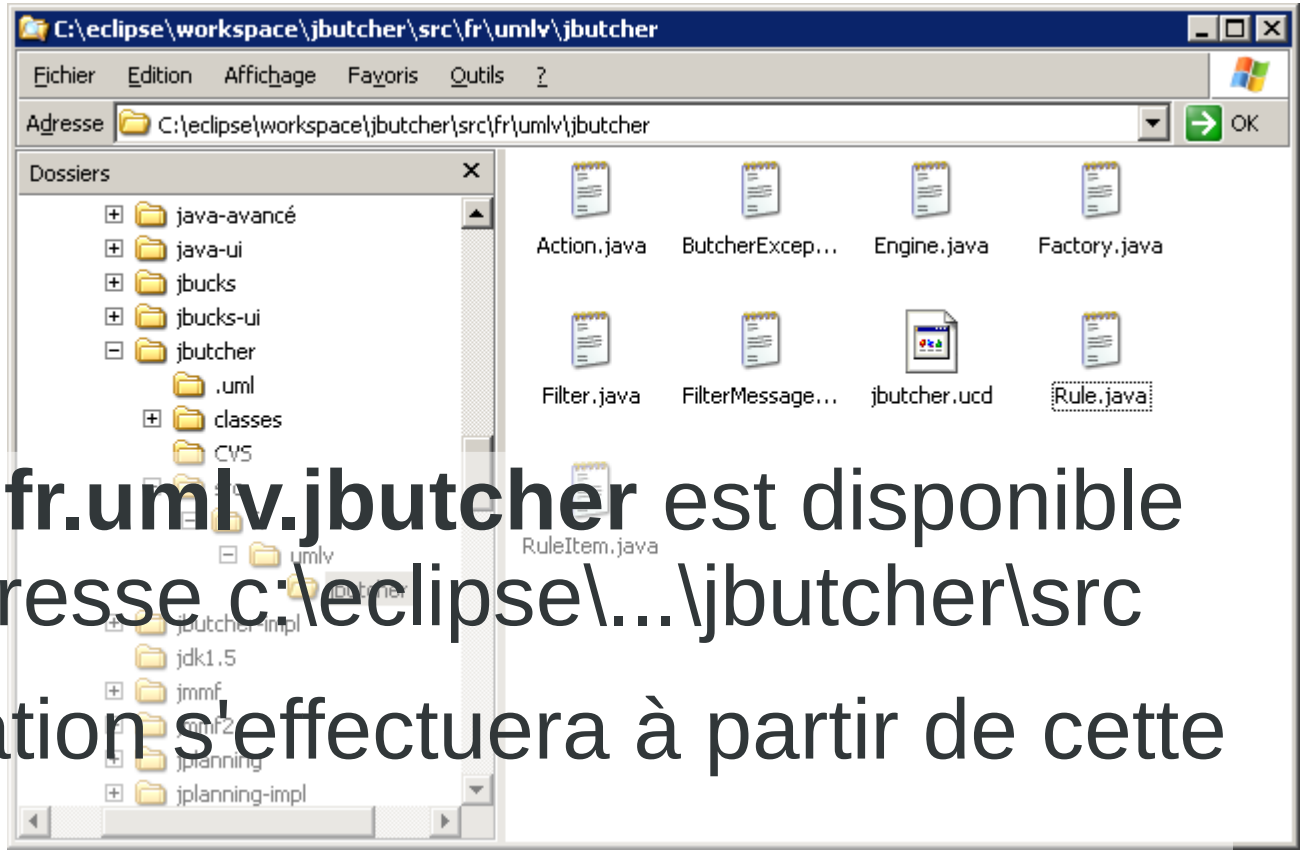
Déclaration de classe avec paquetage :

```
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

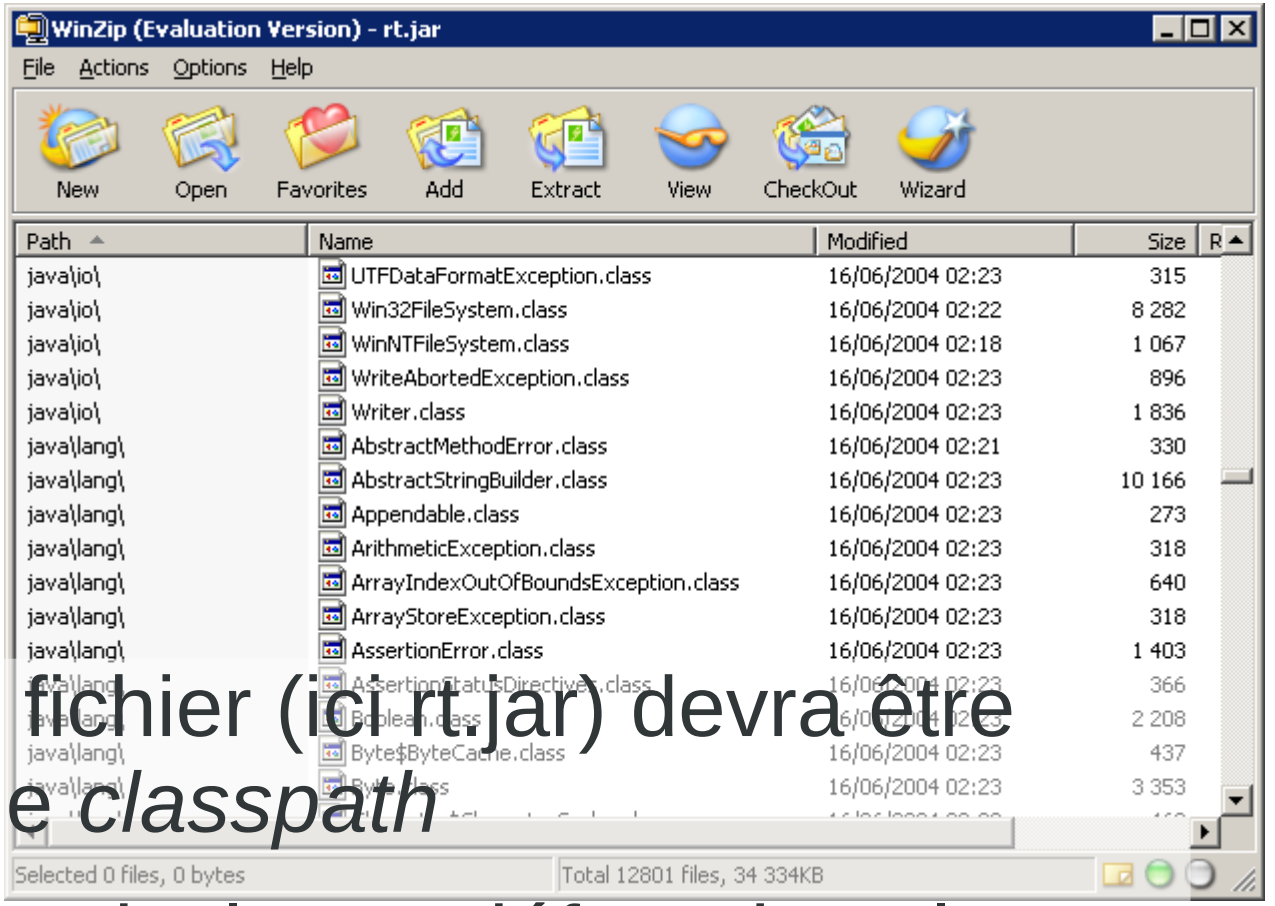


Le paquetage **fr.umlv.jbutcher** est disponible à partir de l'adresse c:\eclipse\...\jbutcher\src

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**

L'instruction **import** permet d'éviter de nommer une classe avec son paquetage

```
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**

La directive **import**

Import est juste une façon de créer **un alias** entre le nom longet le nom court.

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

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

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

ici, le bytecode généré est 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, il y a ambiguïté !

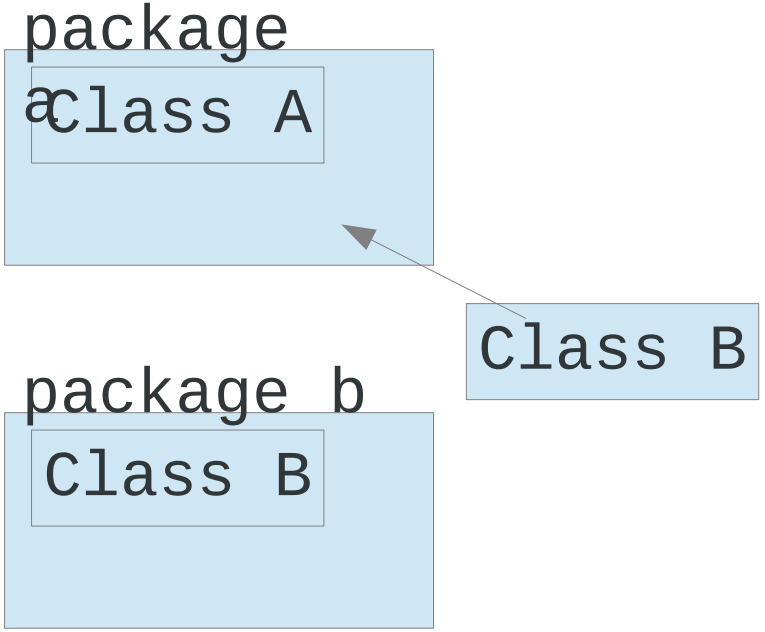
```
import java.util.*;
import java.awt.*;

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

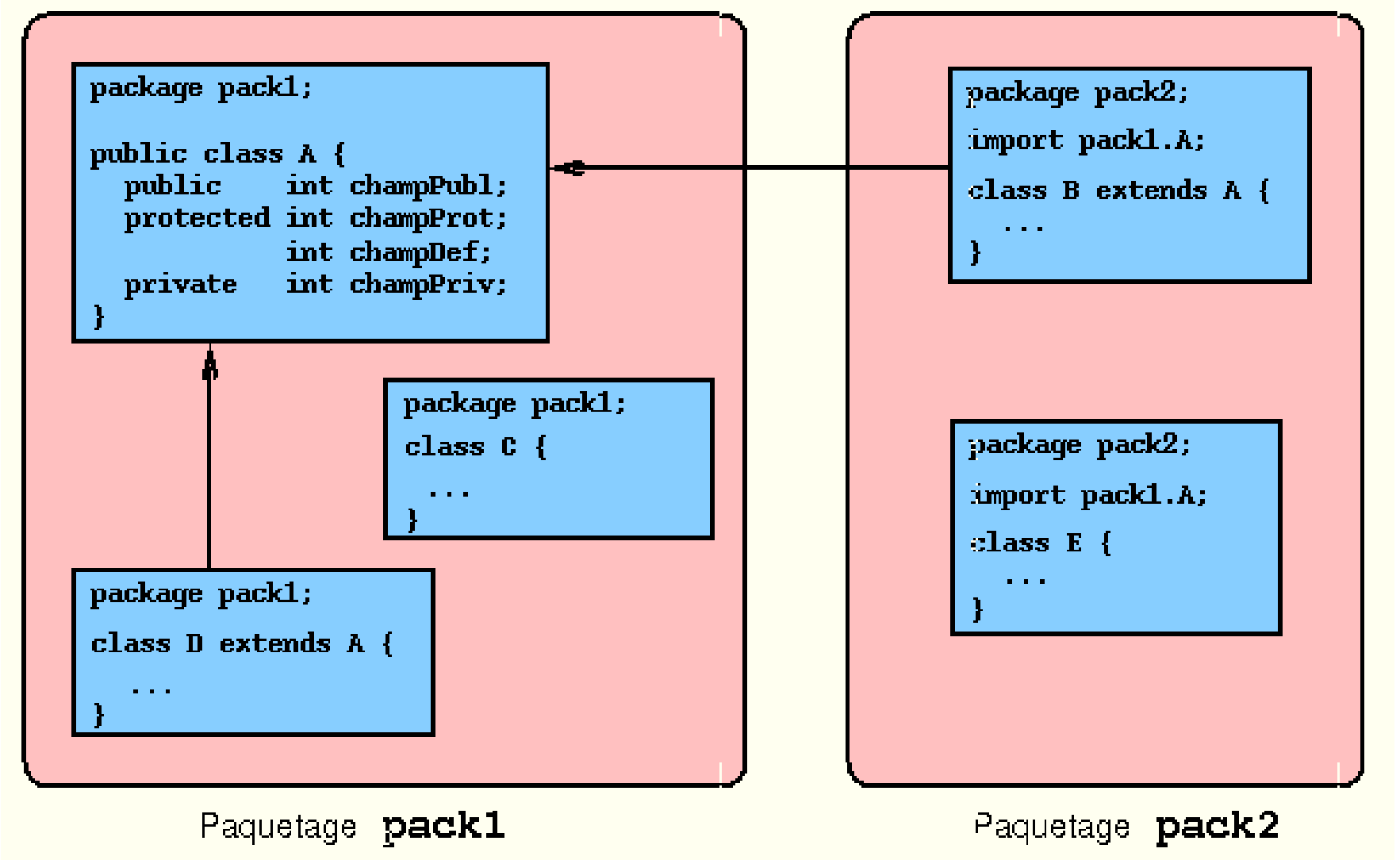
- Permet d'empêcher l'accès à des membres en fonction de la classe qui déclare le membre et de la classe qui y accède
- Très important :
  - Permet l'encapsulation et la protection des données
  - Permet une bonne maintenance en exposant le moins de choses possibles
- L'accès est vérifié par la VM

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



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.