Les bases du langage **Java**

Thierry Vaira

BTS Avignon
SN-IR

v1.0 - 25 août 2016

À lire ...

- ***Programmer en Java***, C. Delannoy (Eyrolles)
- ***Apprendre Java et C++ en parallèle***, Jean-Bernard Boichat (Eyrolles)
- ***Thinking In Java***, Bruce Eckel, version électronique en ligne (anglais/français)

➪ Un grand merci à **Sébastien Jean** (IUT de Valence - Département Informatique) pour son support de cours.

Genèse de Java

- Langage issu d'un projet de *Sun Microsystems*
- Idée : pouvoir déployer facilement des applications dans des équipements électroniques hétérogènes
- Code **interprété** plutôt que natif ➟ utilisation d'une **machine virtuelle**

Notions de bases

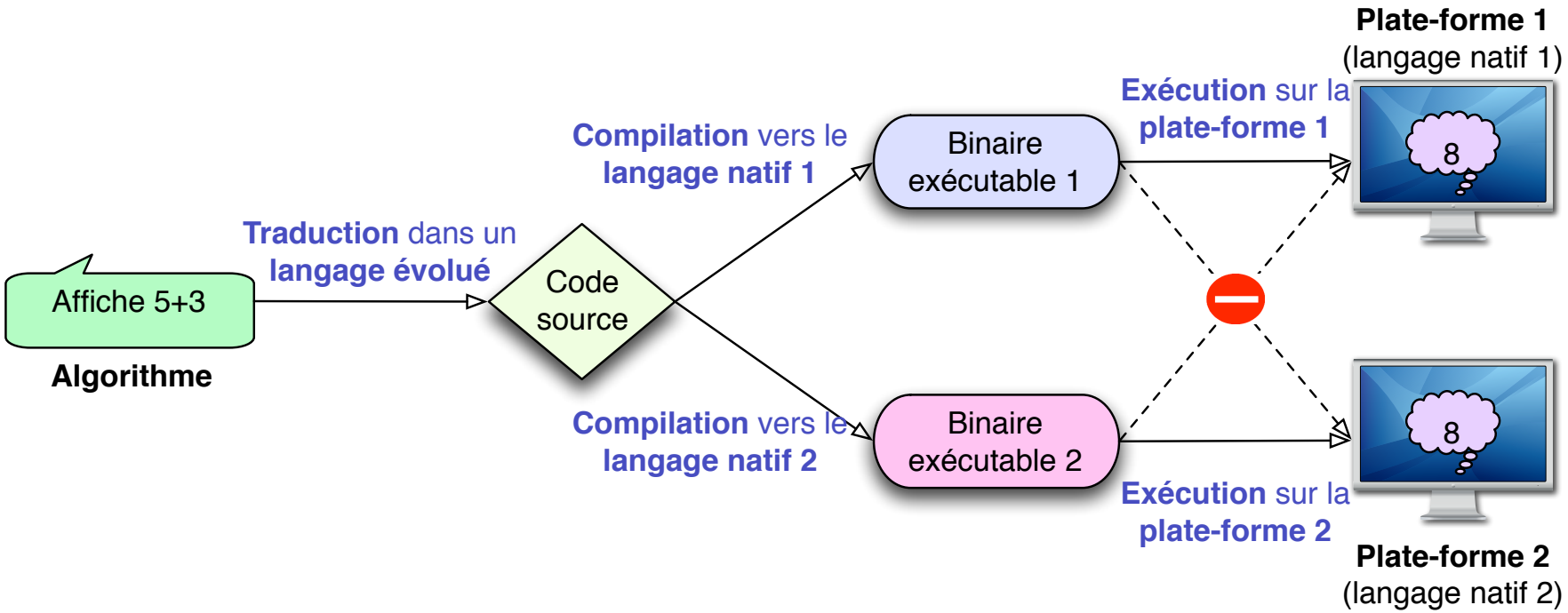
- Java est un **langage orienté objet**
- Il est **interprété** (**bytecode**) et **indépendant de la plate-forme**
- Il est donc **portable**
- Sa **syntaxe** est **simple** (**proche de celle de C++**)

Notions de bases (fin)

- Il offre la **modularité** ➠ Notion de **paquetages**
- Il supporte l'exécution de **tâches concurrentes** ➠ Notion de **fils d'exécution** (***threads***)
- Il est **robuste** et **sûr** ➠ Pas de pointeurs, Notion de **ramasse-miettes** (***garbage collector***)
- Il n'a pas été conçu **pour être rapide**, mais il existe des **techniques d'optimisation** à l'exécution cf. **JIT** (***Just In Time Compiling***)

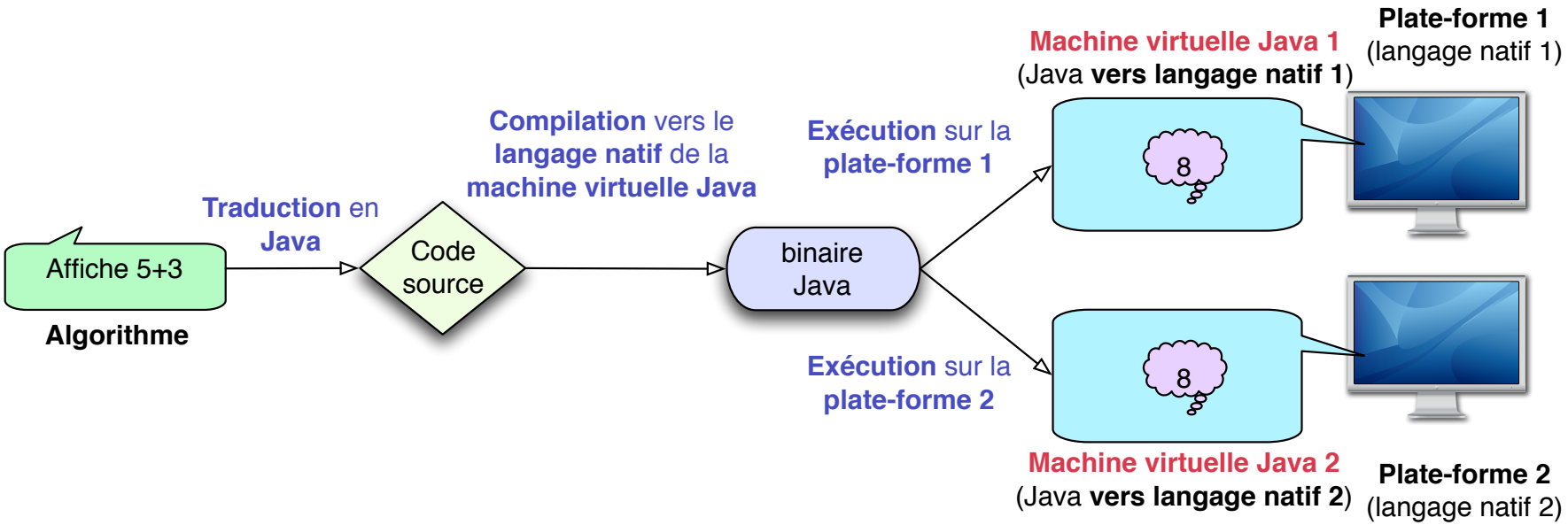
Principe des machines virtuelles

- Chaque **plate-forme matérielle** a sa propre façon de **représenter des nombres**, de les **additionner**, ...
- Pour lui faire **exécuter un programme**, il faut exprimer celui-ci dans son **langage natif**



Principe des machines virtuelles (fin)

- Une **machine virtuelle** est une **application s'exécutant sur une plate-forme réelle et simulant une plate-forme universelle fictive**
- Elle possède un **langage natif propre** et **traduit** un programme écrit dans ce langage vers le langage natif de la plate-forme réelle



Compilation et exécution de programmes Java

Code source Java
(fichier *Helloworld.java*)

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


Compilateur Java

Compilation

Binaire Java
(fichier *Helloworld.class*)

Machine virtuelle Java

Interprétation

Java n'est pas qu'un langage

- **Ensemble de technologies**, de **librairies**, d'**outils**
- **J2SE SDK** (***Java Development Kit***)
  - Outils de base pour le développement
    - Compilateur, machine virtuelle (**JVM**), *Applet Viewer*, désassembleur, ...
  - Bibliothèques standards
    - Notamment les **JFC** (***Java Foundation Classes***)
- **J2SE JRE** (***Java Runtime Environment***)
  - Seulement l'environnement d'exécution

Bibliothèques standards

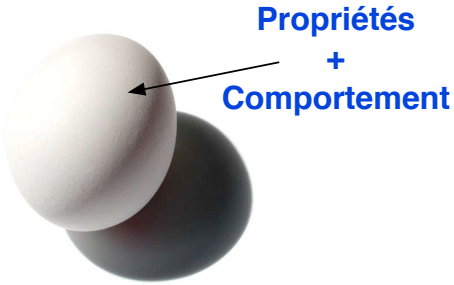
- **JDBC** (***Java DataBase Connectivity***)
  - Interaction avec des bases de données relationnelles (Oracle, MS Access, mySQL, ...)
- **AWT / Swing**
  - Création d'interfaces graphiques
- **JMF** (***Java Media Framework***)
  - Librairies multimédias
- **JAXP** (***Java API for XML Parsing***)
  - Manipulation de documents XML
- **JNI** (***Java Native Interface***)
  - Interaction avec des programmes natifs
- ...

Java à grande échelle

- **J2EE** (***Java 2 Enterprise Edition***)
  - Cible les architectures **N-Tiers**, Web ou intranet
  - « Scripting » client (***applets***), « scripting » serveur (***servlets***) et composants métiers (***EJB***, *Enterprise JavaBeans*)

Qu'est ce qu'un objet ?

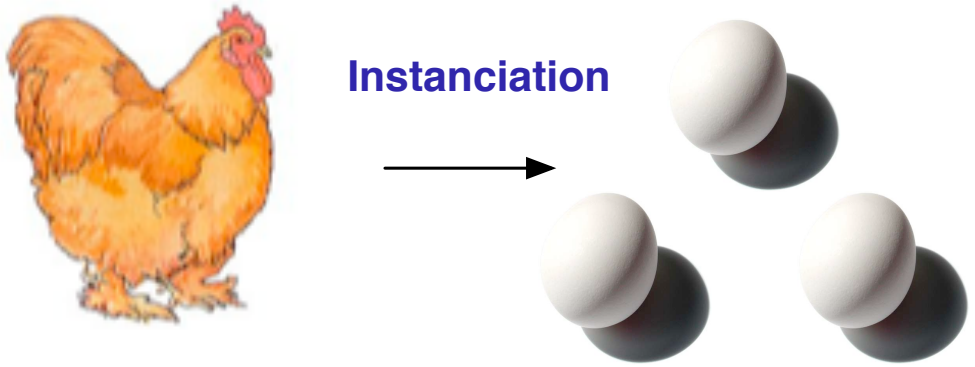
- Un **objet** est caractérisé par le rassemblement, au sein d'une même **unité d'exécution**, d'un ensemble de **propriétés** (constituant son **état**) et d'un **comportement**
  - On parle d'**encapsulation**



- La notion de **propriété** est matérialisée par un **attribut**
  - Comparable à une « *variable locale à un objet* »
- La notion de **comportement** est matérialisée par un ensemble de **méthodes**
  - Comparables à des « *fonctions* » ou à des « *procédures* », actions pouvant être effectuées sur l'objet

Qu'est ce qu'une classe ?

- Une **classe** est une **unité de compilation**, définissant un **type**, qui peut être vue comme un « **moule** »ou une « **fabrique d'objet** »et qui comporte :
  - La **définition des attibuts**
  - La **définition** et l'**implémentation** des **méthodes**
- La « fabrication »d'un objet s'appelle l'**instanciation**
  - L'objet est appelé **instance de classe** et possède le type que définit sa classe
- Remarque : *La classe n'existe qu'à la compilation, l'objet n'existe qu'à l'exécution*



Comment interagir avec un objet ?

- **Seul un objet** (A) peut **interagir avec un autre objet** (B) (**ce peut être lui-même !**)
- Le **code d'une des méthodes de A** contient soit :
  - Un **accès en lecture/écriture** à un **attribut** de B (ou de A)
  - Un **appel** (on parle d'**invocation**) d'une des **méthodes** de B (ou de A)
- Remarque : Les objets interagissent entre eux en s'échangeant des **messages**. La réponse à la réception d'un message par un objet est appelée une **méthode**. Une **méthode est donc la mise en oeuvre du message** : elle décrit la réponse qui doit être donnée au message.

Conception d'une classe : exemple d'une lampe

- Une lampe est **caractérisée par** :
  - Sa **puissance** (une **propriété** ➠ un **attribut**)
  - Le **fait qu'elle soit allumée ou éteinte** (un **état**)
- Au niveau **comportement**, les **actions possibles** sur une lampe sont donc :
  - L'**allumer** (une **méthode**)
  - L'**éteindre** (une autre **méthode**)

Conception d'une classe (suite)

```
public class Lampe
{
   public int puissance;

   public boolean estAllumee;

   public void allumer()
   {
      this.estAllumee = true;
   }

   public void eteindre()
   {
      this.estAllumee = false;
   }

}
```

Conception d'une classe (fin)

Le nom du **fichier** doit **porter le nom de la classe** (en respectant les majuscules) et l'extension ***.java***
Un fichier par classe

```
public class Lampe
{
    public int puissance;

    public boolean estAllumee;

    public void allumer()
    {
        this.estAllumee = true;
    }

    public void eteindre()
    {
        this.estAllumee = false;
    }
}
```

Déclaration d'une classe (et d'un type)

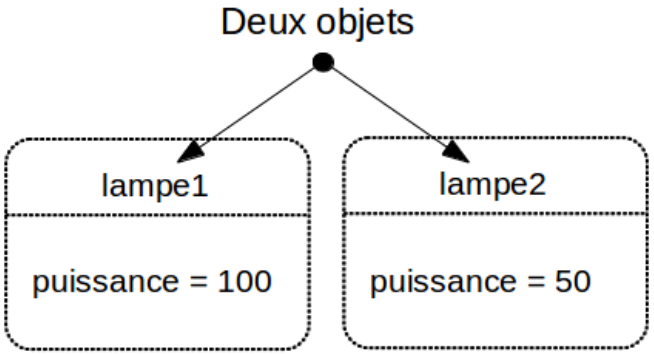
Type

Signature de méthode

Implémentation de méthode

Instances de classe

- Chaque instance possède un **exemplaire propre** de **chaque attribut défini par la classe** (l'état de deux instances est indépendant)



- Le **code des méthodes** est **commun à toutes les instances** (il n'existe qu'une seule fois en mémoire)
- Le mot-clé `this`, utilisé dans le code des méthodes d'une classe, fait **référence à l'instance (l'objet lui-même) sur laquelle est invoquée la méthode**

Types Java

- Les types s'appliquent aux **attributs**, aux **variables locales des méthodes**, aux **paramètres et valeurs de retour des méthodes**
- Il existe **3 catégories de types** en Java
  - Les types **primitifs**
    - Ne sont pas définis par des classes (ne sont donc pas des objets), et ont une représentation immédiate en mémoire
  - Les types **objets**
  - Le type **tableau**
    - Ni un type primitif, ni vraiment un type objet non plus !

Types primitifs

- Types **entiers**

Type	Taille (en bits)	Valeur minimale	Valeur maximale
byte	8	-128	127
short	16	-32768	32767
int	32	-2147483648	2147483647
long	64	-9223372036854775808	9223372036854775807

  - Remarque : les entier peuvent aussi être exprimés en octal (017 pour 15) ou en hexadécimal (0x0F pour 15)

- Types **flottants**

Type	Taille (en bits)	Chiffres significatifs	Valeur minimale	Valeur maximale
float	32	7	1.4239846E-45	3.40282347E38
double	64	15	4.9406564584124654E-234	1.7976931348623 16E308

  - Remarque : notation 3.25 ou -3.23E-12

Types primitifs (fin)

- Type **vide**
  - `void` uniquement pour indiquer qu'une méthode ne retourne pas de valeur
- Type **booléen**
  - `boolean`, possède 2 valeurs : `true` et `false`
- Type **caractère**
  - `char`, valeur entre apostrophes (ex : `'a'`), représente un caractère Unicode (16 bits)
  - Les caractères sont des codes, on peut utiliser des opérations arithmétiques sur les caractères (`'a' + 1 = 'b'`)
  - `\uxxxx` permet de représenter le caractère Unicode de code `xxxx` si la plateforme n'en permet pas l'édition

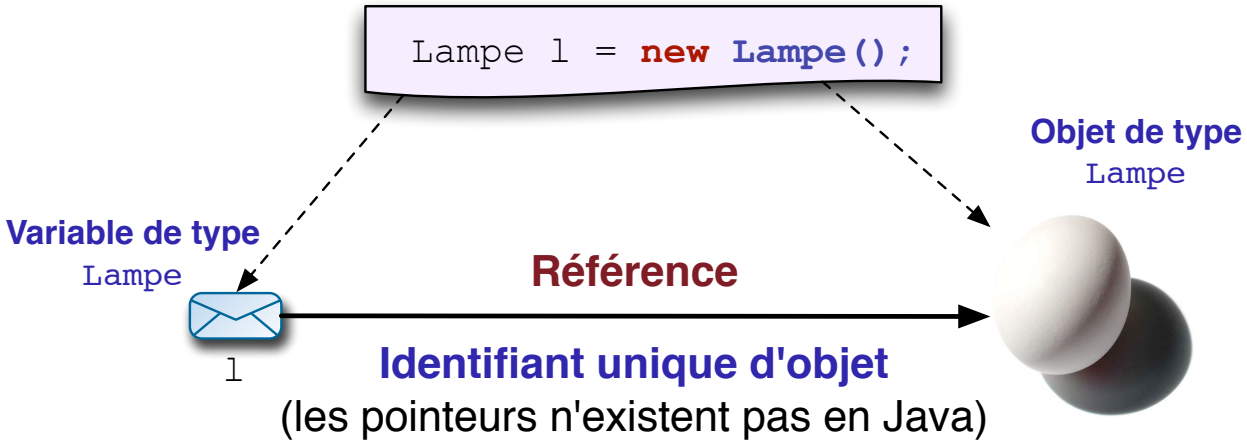
Instanciation d'un objet

- L'**instanciation d'un objet** est réalisée, **n'importe quand**, par **appel à un constructeur** (porte le même nom que la classe)
  - Mot clé `new`

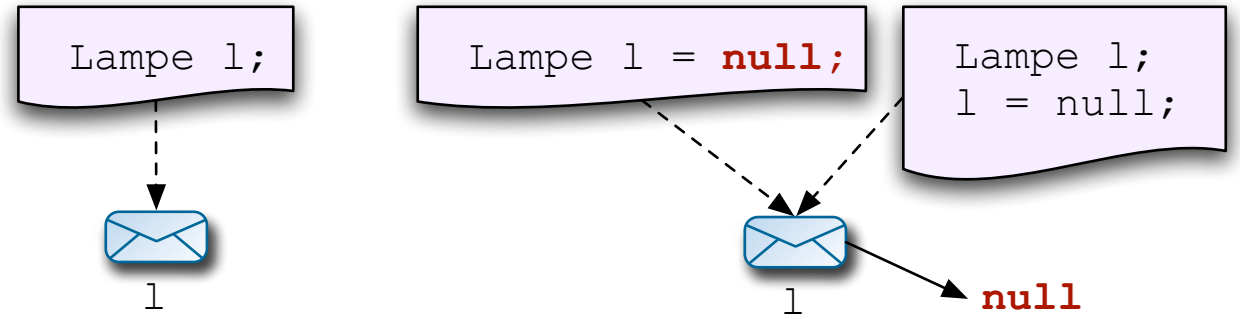
```
Lampe l = new Lampe();
```

- La **variable** `l`, de **type** Lampe, n'est pas un objet Lampe !
- La variable `l` est une **référence** sur un **objet de type** Lampe
- ***Analogie avec C :***
  - Un objet de type Lampe est analogue à une structure de données en mémoire
  - Une variable de type Lampe est analogue à un pointeur sur une structure

Instanciation d'un objet (fin)



- **Attention aux variables non initialisées !**



Constructeur

- D'un point de vue **système**, le constructeur a pour but d'**allouer un nouvel objet en mémoire**

- D'un point de vue **fonctionnel**, le constructeur a pour but de **créer un nouvel objet, dans un état « utilisable »**
  - Il doit contenir les instructions d'**initialisation des attributs de l'objet**

```
public class Lampe
{
    ...
                                    Constructeur
    public Lampe()
    {
      this.puissance = 100;
      this.estAllumee = false;
    }

}
```

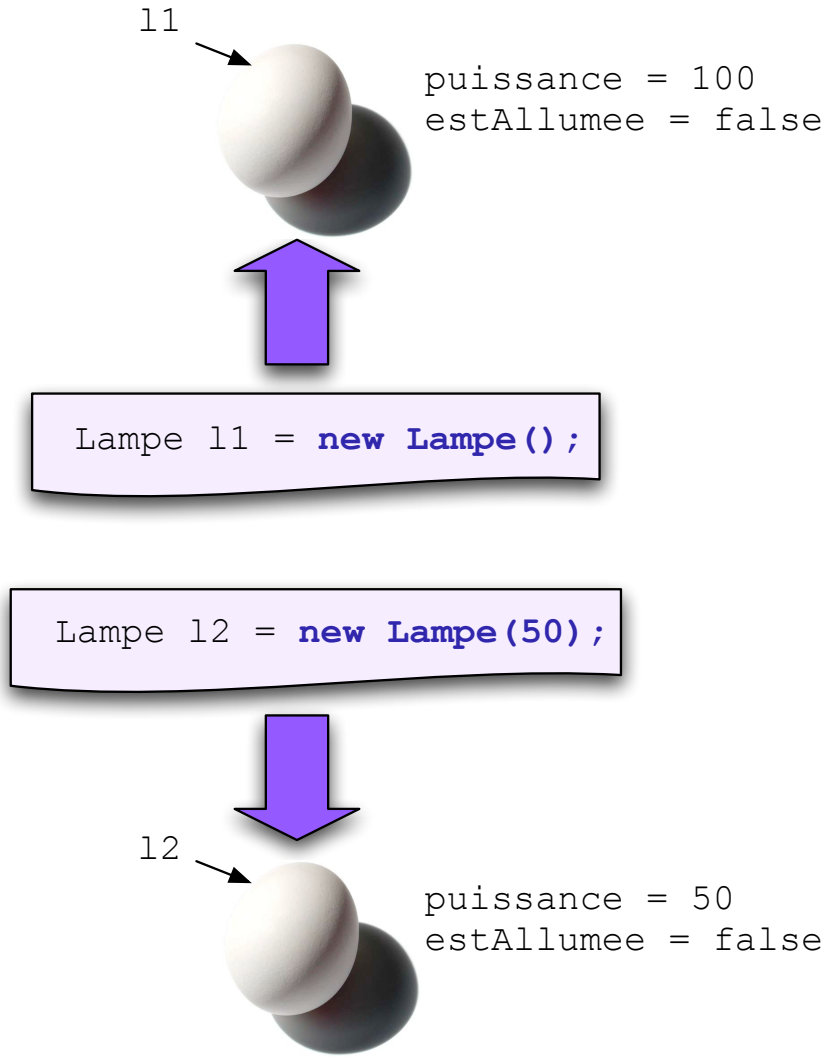
Constructeur (suite)

- Il existe **implicitement** un **constructeur par défaut**
  - Il se contente d'**allouer l'objet sans initialiser aucun attribut**
    - Il est **fortement recommandé** d'initialiser tous les attributs, Il existe des initialisations par défaut qui ne sont pas toujours souhaitables
  - Il est **sans paramètre**
  - **Il n'existe plus si un autre constructeur est défini**
- Il est possible de définir **plusieurs constructeurs** avec des **paramètres différents**

Constructeur (fin)

- Et la puissance dans tout ça ?

```
public class Lampe
{
    ...

    public Lampe()
    {
      this.puissance = 100;
      this.estAllumee = false;
    }

    public Lampe(int p)
    {
      this.puissance = p;
      this.estAllumee = false;
    }

}
```



Accès aux attributs, appels de méthodes

- **Notation pointée : *Référence d'objet.attribut ou méthode***
- Accès aux attributs

```
int p1 = l1.puissance;
```

- Appel de méthode

```
if (l2.estAllumee) l2.eteindre();
else l2.allumer();
```

Accesseurs, modificateurs d'accès

- Fixer la puissance une fois pour toutes ?

```
public class Lampe
{
   public int puissance;
   ...

   public Lampe(int p) {...}

   ...
}
```

```
Lampe l = new Lampe(40);
l.puissance = 150;
```

Problème !

- Il est toutefois judicieux de **ne pas exposer les attributs**, en les rendants **privés**, et permettre un accès extérieur en lecteur et/ou écriture via des méthodes appelées **accesseurs**

```
public class Lampe
{
   private int puissance;
   ...

   public Lampe(int p) {...}

   public int getPuissance()
   {return this.puissance;}

   ...
}
```

```
Lampe l = new Lampe(40);
l.puissance = 150;  Interdit !
int p = l.getPuissance();
```

Constantes

- Déclaration de constante

```
public class Lampe
{
   public final static int P_MAX = 150;

   ...
}
```

- Utilisation

```
int p = ...;

if (p <= Lampe.P_MAX)
{
   Lampe l = new Lampe(p);
   ...
}
```

Exécution d'une application Java

- Une **application Java** peut être vue comme un **ensemble d'objets traversés par un flot d'exécution**
- L'exécution d'une application Java (mono-tâche) est une **séquence d'envois de messages d'objet à objet** : appels de méthodes, accès à des attributs
- Une **application Java** est définie par un ensemble de classes dont une possède une méthode `main` possédant la signature (exacte) suivante :

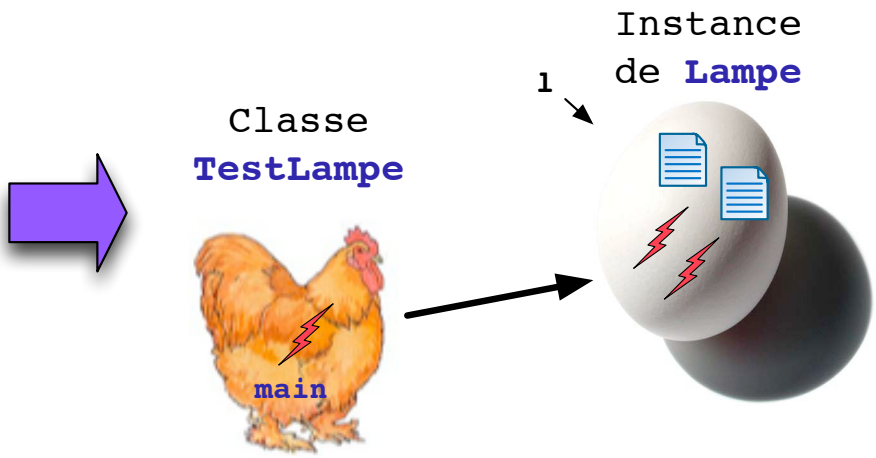
```
public static void main(String[] args);
```

- Comme en C, **l'exécution d'une application Java débute par un appel à `main`**

Exécution d'une application Java (fin)

- En règle générale, la méthode `main` est déclarée dans une **classe à part et ne définissant pas de type**

```
public class TestLampe
{
   public static void main(String[] args)
   {
      Lampe l = new Lampe();
      l.allumer();
   }
}
```



- Pour exécuter cette application, la machine virtuelle Java **« charge »la classe** `TestLampe`, puis exécute le `main`.
Le chargement des autres classes s'effectue lors de leur première utilisation. L'édition de lien s'effectue au fur et à mesure de l'exécution.

Déclaration de variables, affectation

- Les variables peuvent être déclarées **n'importe où dans un bloc de code** ({...})
- **Une variable de type primitif** contient une **valeur immédiate**
- **Une variable de type tableau ou objet** contient une **référence**

```
public static void main(String[] args)
{
    float c;

    int i = 0;

    Lampe l ;

    c = 3; // une instruction

    // notez au passage la syntaxe d'un
    commentaire de fin de ligne ...
    /* ceci est un autre commentaire qui
    n'invalide pas le reste de la ligne... */

    boolean b = (c > i);

    // affiche la valeur de b sur la console
    et passe a la ligne
    System.out.println(b);}
```

Déclaration de variables, affectation (fin)

- Lors de la **déclaration d'un tableau**, **on ne spécifie pas la taille**
- **Un tableau n'existe qu'après l'appel au constructeur !**

```
public static void main(String[] args)
{
    // args représente les arguments passés à
    l'application sur la ligne de commande.
    args[0] est le premier "vrai" argument.

    // Déclaration d'un tableau statique de 2
    valeurs de type short (espace mémoire fixe)
    short[] s = new short[2];

    // Les indices varient de 0 à taille-1
    s[0] = 3;
    s[1] = s.length; // les tableaux possèdent un
    attribut length qui donne leur capacité

    // On ne peut pas afficher tel quel un
    tableau,l'argument de println est une chaîne de
    caractères. Java convertissant automatiquement
    les types, on peut écrire :

    System.out.println(s[0]+""+s[1]);
}
```

Surcharge de méthode

- Il est possible en Java de **définir plusieurs méthodes portant le même nom** mais uniquement avec des **signatures différentes** : on parle de **méthode surchargée**
  - Le **nombre**, l'**ordre** ou les **types des paramètres doivent être différents**
    - `public void methode() {...}`
    - `public void methode(boolean o) {...}`
    - `public int methode(int d) {...}`
    - `public int methode() {...}`
      Interdit car mêmes paramètres que la première

Méthodes et attributs statiques

- Un **attribut statique** est un attribut qui n'existe qu'en **un unique exemplaire** et qui est **partagé par toutes les instances de la classe** : **pas de copie locale**
- Déclaration :

```
private static int lampes;
```

- Accès :

```
int i = Lampe.lampes;
```

- Une **méthode statique**, ou *méthode de classe*, n'est **pas associée à une instance particulière mais à la classe elle-même**
  - Elle reflète en général un **comportement global**, comme par exemple un compteur d'instances

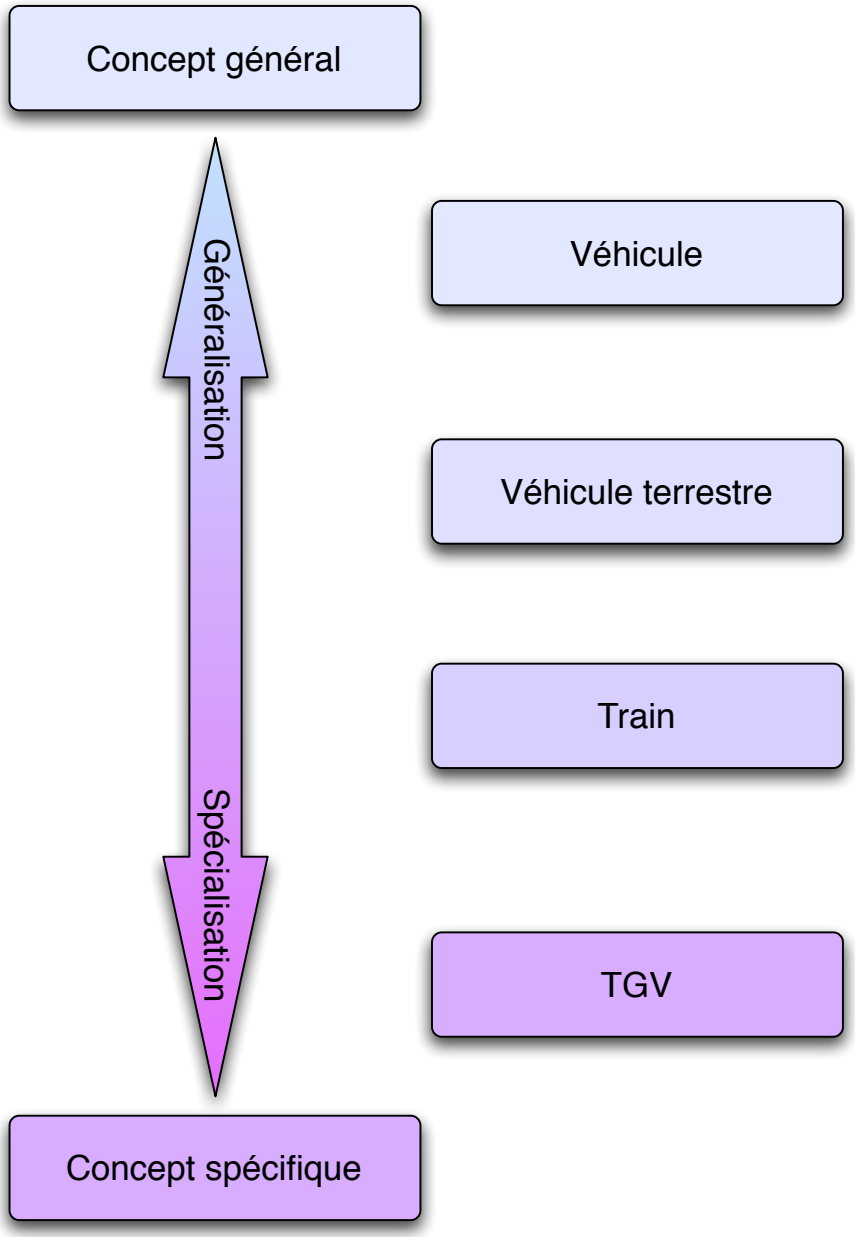
```
public static int getLampes()
{return Lampe.lampes};
```

Méthodes et attributs statiques (fin)

- Exemple : un compteur d'instances

```
public class Lampe
{
   private static int lampes = 0;
   ...
   public Lampe(int p)
   {
      this.puissance = p;
      this.estAllumee = false;
      Lampe.lampes++;
   }

   public static int getLampes()
   {return Lampe.lampes;}
   ...
}
```

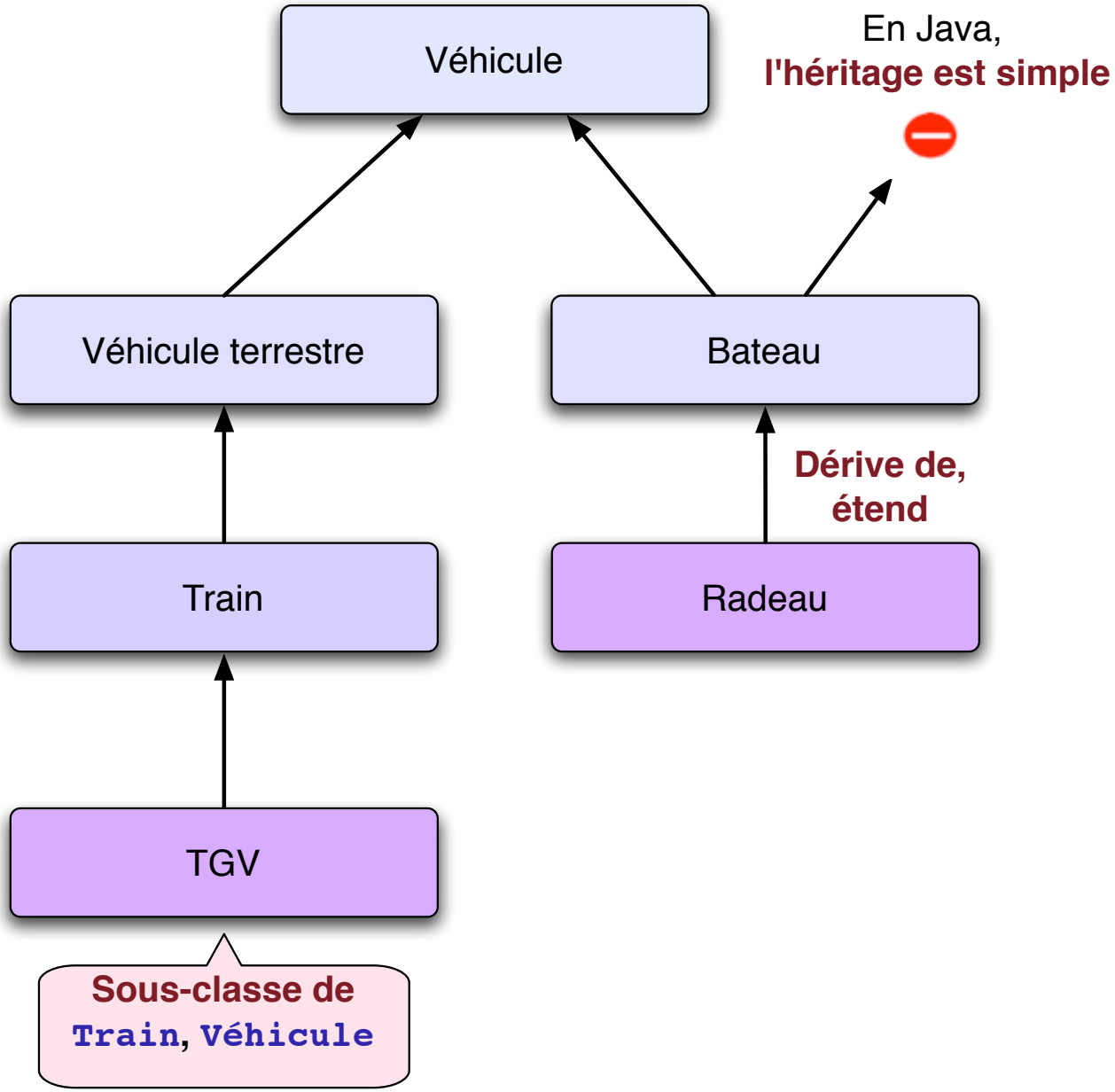
Héritage : introduction



Héritage : introduction (suite)

- L'**héritage** permet la **réutilisation** en **spécialisant** des **concepts plus abstraits**
- L'héritage consiste à **affiner la définition d'un concept** en :
  - **Ajoutant** des **propriétés**
  - **Ajoutant** des éléments de **comportement**
  - **Redéfinissant** des éléments de **comportement existants**
- En Java, on appelle :
  - **Classe de base**, ou **super-classe**, la classe **plus abstraite**
  - **Classe dérivée**, ou **sous-classe**, la classe **plus spécifique**

Héritage : introduction (fin)



Prenez garde à la fermeture des portes . . .

```
public class Wagon
{
   private Siege[] sieges;
   private int classe;
   private int numero;

   public Wagon(int classe, int nbSieges)
   {
      this.numero = 0;
      this.classe = classe;
      this.sieges = new Siege[nbSieges];
      for (int i=0;i<nbsieges;i++)
         this.sieges[i] = new Siege(i, false);
   }

   public void affecterNumero(int numero) {...}

   public void occuperSiege(int numero) {...}

   public void libererSiege(int numero) {...}
}
```

→ **Aggrégation**

```
public class Siege
{
   private int numero;
   private boolean occupe;

   public Siege(int numero, boolean occupe)
   {
      this.numero = numero;
      this.occupe = occupe;
   }

   public int getNumero() {...}

   public boolean estOccupe() {...}

   public void occuper() {...}

   public void liberer() {...}
}
```

Nous vous informons qu'un bar [1] est situé en voiture 5

```
public class WagonBar extends Wagon
{
   private Bar bar;
```

Ajout de propriétés

Constructeur

Ajout d'éléments de comportement

```
   public Boisson commanderBoisson(int codeBoisson)
   {...}

   public EnCas commanderEnCas(int codeEnCas)
   {...}
}
```

1. L'abus d'alcool est dangereux pour la santé.

Relations entre classe et sous-classe

- La **classe dérivée** **hérite** des **attributs** et **méthodes** de sa **super-classe**
  - Un `WagonBar` **est** (avant tout) **un** `Wagon`
  - Remarque : **la super-classe peut elle-même hériter d'une autre classe**
  - Accès aux attributs et méthodes **visibles** avec le mot-clé **this**, depuis le code de la classe dérivée

Modificateurs d'accès

- **public**
  - Accessible depuis le code de **toutes les classes**
- **private**
  - Accessible uniquement depuis le code de **la classe elle-même**
- **protected**
  - Accessible depuis la **classe**, ses **sous-classes** et les **classes du même *paquetage***
- ***[sans modificateur]***
  - Accessible depuis la **classe** et les **classes du même paquetage**

Modificateurs d'accès (fin)

```java
public class A
{
   public int a;

   private int b;

   ...

   public void faireUnTrucAveca() {...}
   private void faireUnTrucAvecb() {...}
   protected void faireUnAutreTrucAvecb() {...}
}
```

```java
public class B extends A
{
   public int c;

   public void uneMethode()
   {
     this.a = 3;
     this.b = 4;
     this.faireUnTrucAveca();
     this.faireUnTrucAvecb();
     this.faireUnAutreTrucAvecb();
   }
}
```

```java
public class C
{
   public A attr;

   public void uneMethode()
   {
     this.attr.a = 3;
     this.attr.b = 4;
     this.attr.faireUnTrucAveca();
     this.attr.faireUnTrucAvecb();
     this.attr.faireUnAutreTrucAvecb();
   }
}
```

Classe Wagon et modificateurs d'accès

```
public class Wagon
{
  private Siege[] sieges;
  private int classe;
  private int numero;

  public Wagon(int classe, int nbSieges) {...}

  public void affecterNumero(int numero) {...}
  public void occuperSiege(int numero) {...}
  public void libererSiege(int numero) {...}
}
```

Inaccessible depuis WagonBar

Accessible depuis WagonBar

Héritage et constructeurs

- Le **constructeur de la classe dérivée** doit **intégralement** prendre en charge la **construction de l'objet**
  - Il peut s'appuyer sur le **constructeur de la classe de base**, en y faisant **appel**
  - L'**appel au constructeur de la super-classe** doit être la **première instruction du constructeur de la sous-classe**
  - L'appel au super-constructeur s'effectue via le mot-clé `super`
    - `super()` pour l'appel au constructeur sans paramètre
    - `super(a,b)` pour l'appel au constructeur avec 2 paramètres
    - Si aucun appel n'est explicité, le compilateur ajoute un **appel implicite au super-constructeur sans paramètre s'il existe**
  - Attention ! `super` désigne uniquement la **classe immédiatement supérieure ! ! !**

Constructeur de la classe WagonBar

```
public class WagonBar extends Wagon
{
   private Bar bar;

   public WagonBar(int classe, int nbSieges, Bar bar)
   {
      super(classe, nbSieges);
      this.bar = bar;
   }

   public Boisson commanderBoisson(int codeBoisson)
   {...}

   public EnCas commanderEnCas(int codeEnCas)
   {...}

}
```

Paramètres généraux + spécifiques

Redéfinition de méthode

- **Une partie du comportement existant d'une classe peut être redéfinie dans une classe dérivée**
  - **Redéfinition de méthode**
- Une **méthode redéfinie** possède **obligatoirement la même signature** que la méthode de la super-classe (sinon, c'est une **surcharge**)
  - Mêmes paramètres, même type de retour
  - **Le modificateur d'accès ne doit pas être plus restrictif**
- Remarque : il n'est **pas possible** de **redéfinir une méthode statique**

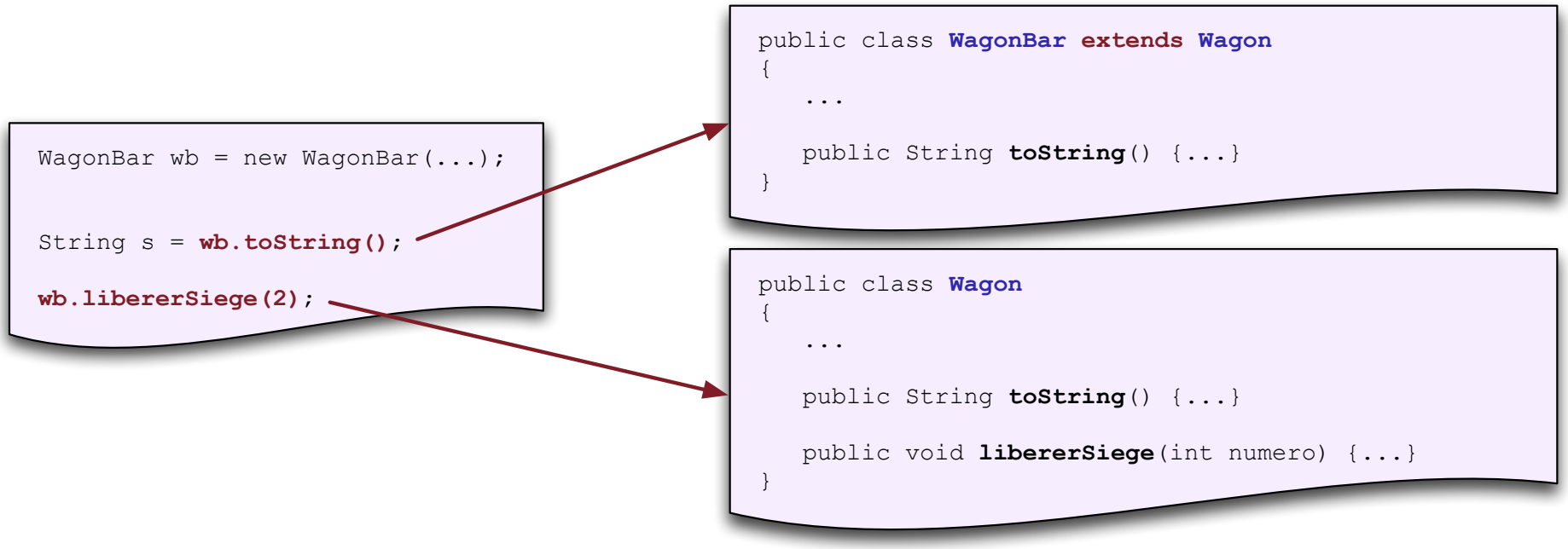
Redéfinition de méthode (suite)

```
public class Wagon
{
   ...

   public String toString()
   {
      return  "Wagon de classe"+this.classe
             +", n° "+this.numero
             +", "+this.sieges.length+" sieges";
}
```

```
public class WagonBar extends Wagon
{
   ...

   public String toString()
   {
      
   }
}
```

Redéfinition de méthode (suite)

- Invocation de méthode sur un objet : **liaison tardive** ou ***lookup***
  - **Lorsqu'une méthode est invoquée sur un objet**, la **méthode exécutée** est la **première méthode accessible conforme à la signature** en **cherchant à partir de la classe** et en **remontant** l'**arbre d'héritage**



Redéfinition de méthode (fin)

- Le mot-clé `super` : lookup à partir de la **super-classe**
  - `this` représente l'**objet courant** avec le **type associé à sa classe**
  - `super` représente l'**objet courant** avec le **type associé à sa super-classe**

```
public class WagonBar extends Wagon
{
   ...

   public String toString()
   {
      return super.toString() + ", avec bar";
   }
}
```

Classes et méthodes finales

- Le modificateur d'accès `final` empêche
  - La **redéfinition** lorsqu'il est appliqué à une **méthode**

```
public class Wagon
{
   ...

   public final String toString() {...}
   public void libererSiege(int numero) {...}
}
```

```
public class WagonBar extends Wagon
{
   ...

   public String toString() {...}
}
```

  - La **dérivation** lorsqu'il est appliqué à une **classe**

```
public final class Wagon
{
   ...

   public String toString() {...}
   public void libererSiege(int numero) {...}
}
```

```
public class WagonBar extends Wagon
{
   ...
}
```

Polymorphisme

- **Polymorphisme**
  - Un `WagonBar` peut **« être considéré comme »** un `Wagon`
- Basé sur la **compatibilité de type**
  - `WagonBar` est compatible avec `Wagon` mais pas l'inverse !

```
Wagon w = new Wagon(...);
WagonBar wb = new WagonBar(...);

w = wb;  Autorisé, un WagonBar est un Wagon

wb = w; Interdit, un Wagon n'est pas un WagonBar
```

Polymorphisme (suite)

- **Attention ! L'affectation d'un type compatible n'est pas un forçage de type !**
- Dans l'exemple précédent, les règles qui s'appliquent sont les suivantes :
  - Les méthodes et attributs **accessibles** sont ceux de `Wagon`
  - Si des méthodes de `Wagon` sont redéfinies dans `WagonBar`, ce sont celles-ci qui seront invoquées

```
Wagon w = new Wagon(...);
WagonBar wb = new WagonBar(...);

w = wb;

System.out.println(w.toString());
```

WagonBar

Polymorphisme (fin)

- Avec le polymorphisme, on peut manipuler des objets sans forcément (**généricité**)
- Polymorphisme + redéfinition = généricité + **comportement adapté**

```
Wagon[] wagons = new Wagon[2];
Wagon w = new Wagon(...);
WagonBar wb = new WagonBar(...);

wagons[0] = w;
wagons[1] = wb;

for (int i=0;i<wagons.length;i+)
   System.out.println(wagons[i].toString());
```

La classe Object

- La classe `Object` est la **racine de tous les arbres d'héritage**

Méthodes de la classe `java.lang.Object`		
public	Object()	*Constructeur*
public Object	clone()	*Clonage de l'objet. En général redéfinie dans les sous-classes*
public boolean	equals(Object o)	*Comparaison d'objets (égalité de références). En général redéfinie dans les sous-classes*
public final Class	getClass()	*Obtention d'informations liées à la classe associée à l'instance*
public int	hashCode()	*Obtention d'une valeur de hachage*
public String	toString()	*Obtention d'une description de l'objet, en langage naturel. En général redéfinie dans les sous-classes.*

La classe Object (suite)

- Méthode `equals()` de la classe `Object` : deux objets sont identiques s'ils ont la **même référence**
  - **A redéfinir dans les sous-classes en fonction des besoins**

```
Wagon w1 = new Wagon(...);
Wagon w2 = new Wagon(...);

System.out.println(w2.equals(w1));

w2 = w1;
System.out.println(w2.equals(w1));
```

La classe Object (suite)

- La méthode `clone` crée une **copie** de l'objet
  - Renvoie une référence de type `Object` : **générique**
- Il existe deux façon de copier :
  - **Superficiellement** (comportement par défaut de la classe `Object`)
    - Création d'un **nouvel objet**
    - **Copie de la valeur** des **attributs de type primitif**
    - **Copie de référence** des **attributs de type objet ou tableau**
  - **En profondeur**
    - **Copie superficielle avec clonage des objets référencés**

Retour sur la compatibilité de type

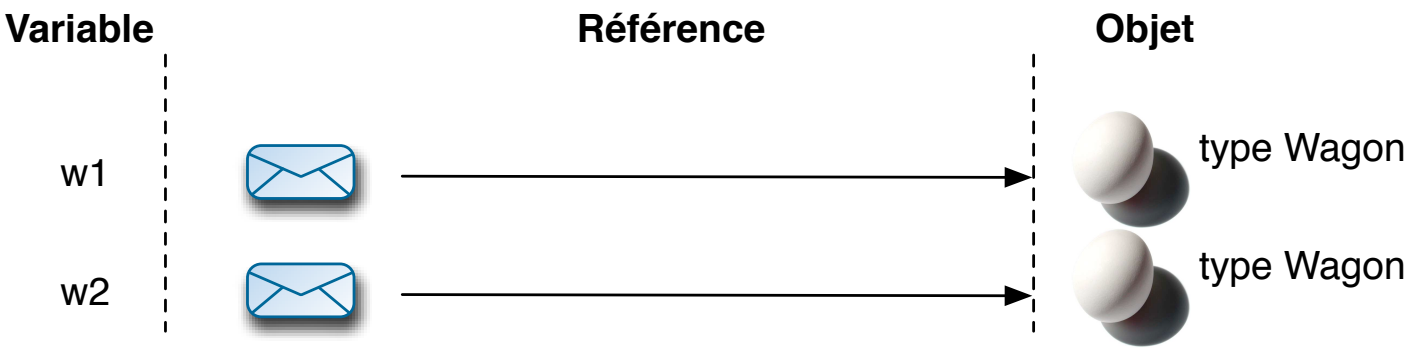
```
Wagon w = new WagonBar(...); Autorisé
WagonBar wb = new WagonBar(...);

Object o = wb.clone(); Autorisé

WagonBar wb2 = wb.clone(); Interdit !

o.libererSiege(1); Interdit !

wb = (WagonBar) w; Autorisé, à condition que l'on
                   force le type et que l'objet soit
                   réellement de type WagonBar
                   (vérifié à l'exécution)
```

La classe Object (fin)

- La méthode `toString` permet d'obtenir une **description textuelle** de l'objet
  - A redéfinir dans les sous-classes en fonction des besoins (cf. `Wagon`)
  - Pour la classe `Object`, retourne une chaîne *classe@hashcode*
    - En s'appuyant sur les méthodes `hashCode` et `getClass`

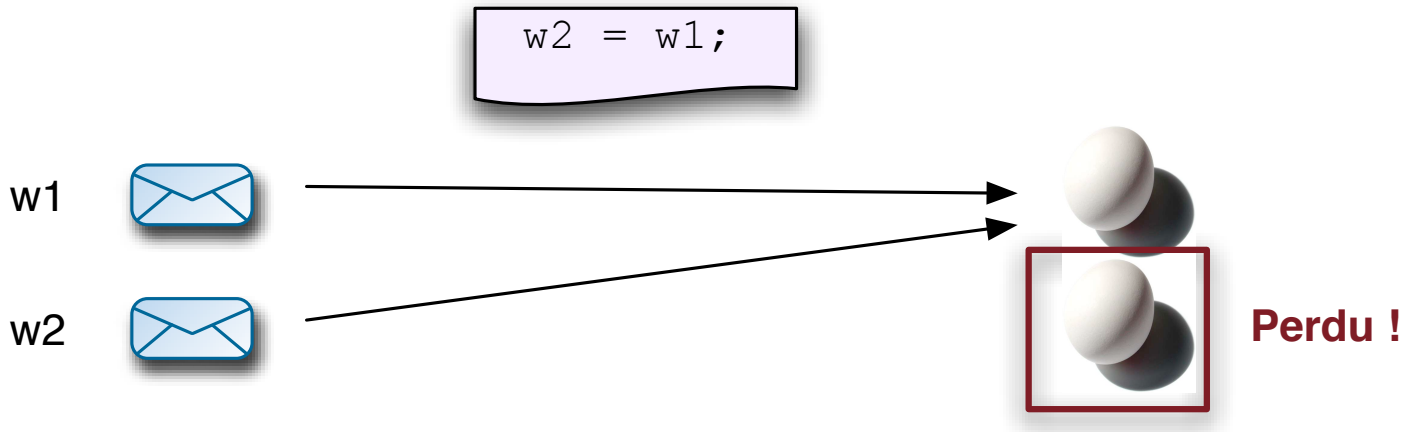
Variables et références

```
Wagon w1 = new Wagon(...);
Wagon w2 = new Wagon(...);
```



- `w1` et `w2` sont des **variables de type** `Wagon` qui contiennent chacune une **référence sur un objet de type** `Wagon`
- La **référence** d'un objet **n'est pas une adresse mémoire**, c'est un **identifiant unique** manipulée par la **machine virtuelle**

Variables et références (fin)

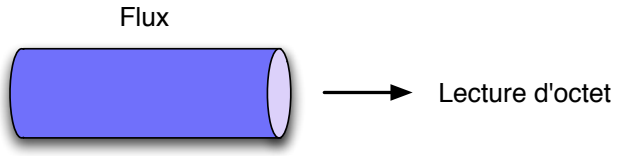


- `w1` et `w2` référencent toutes les deux le même objet
- Lorsqu'il n'existe plus aucune variable référençant un objet, il est **détruit** par un mécanisme appelé **ramasse-miettes** (ou ***Garbage Collector***)

Notion de classe abstraite

- Quelquefois, la **modélisation d'un concept très général** conduit à laisser des **« trous » dans l'implémentation**

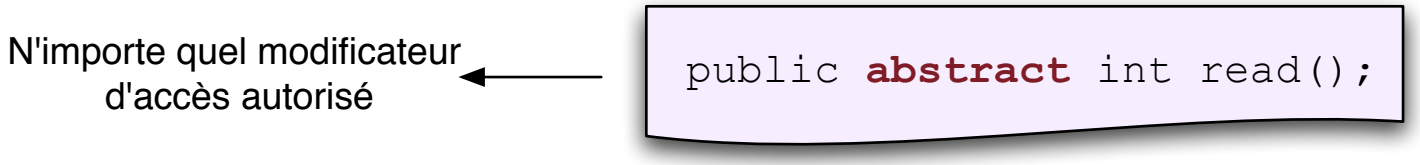
- Exemple d'un flux d'octets en entrée



1. Savoir s'il y a des octets à lire : `int available()`
2. Fermer le flot : `void close()`
3. Lire un octet : `int read()`
4. Remplir un buffer externe : `int read(byte[] b, int o, int l)`
5. « Sauter »n *n* octets : `long skip(long l)`

Notion de classe abstraite (suite)

- Il est possible de **définir une méthode par sa signature uniquement** (**sans en donner l'implémentation**))
  - Cette méthode est appelée **méthode abstraite**



- Une classe définissant au moins une méthode abstraite est dite **abstraite**
  - Il est préférable de le signaler en ajoutant le mot-clé `abstract` avant le mot-clé `class`
- **On ne peut pas instancier d'objet à partir d'une classe abstraite**
  - Mais on le peut à partir d'une **sous-classe** à condition qu'elle **définisse complètement l'implémentation**

Notion de classe abstraite (fin)

```
public class FileInputStream extends InputStream
{
    public FileInputStream(String path) {...}

    ...

    Implémentation de toutes les méthodes abstraites

}
```

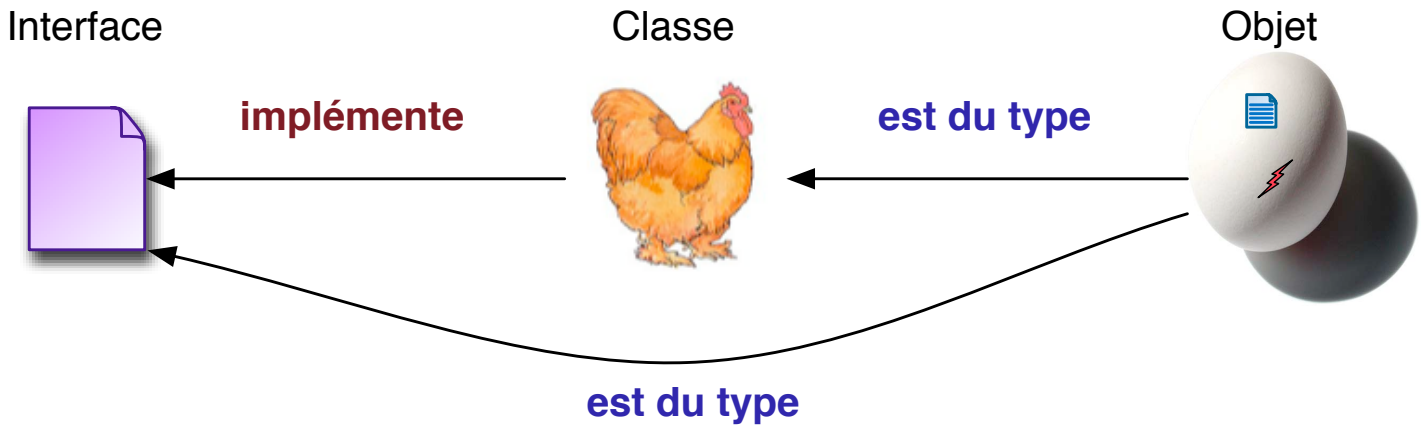
```
public class Test
{
    public static void main (String[] args)
    {
      InputStream is;
      is =  new InputStream(); Interdit !

      FileInputStream fis = new FileInputStream("./readme.txt"); Autorisé

      is = fis; Autorisé

      byte[] b = new byte[1024];
      ...
      is.read(b,0,3); // Lire 3 octets dans le fichier
      fis.close();  // et fermer
}
```

Notion d'interface

- Concept d'**interface** = concept de **classe abstraite** poussé à l'extrême
- Une interface est une **expression de pure conception**, **aucun élément d'implémentation n'est défini**
- Une interface comporte simplement
  - Des déclarations de **constantes**
  - Des **signatures de méthodes** (obligatoirement **publiques**)
- Une interface ne peut pas comporter
  - De déclarations d'**attributs**
  - D' **implémentations de méthodes**

Notion d'interface (suite)

- **On ne peut pas instancier d'objet à partir d'une interface**
  - Mais on le peut à partir d'une **classe** à condition que celle-ci **implémente l'interface**
- Une **classe** exprime une **implémentation**, une **interface** exprime un **contrat**



Notion d'interface (suite)

- On dit qu'une classe **implémente** une **interface si et seulement si** elle **définit l'implémentation de toutes les méthodes de l'interface**
  - Les méthodes doivent avoir **strictement la même signature**

```
public interface Consommateur
{
   public void consomme(Object produit);
}
```

```
public class ConsommateurDeChaines
implements Consommateur
{
   public void consomme(Object o)
   {
      System.out.println(o);
   }
}
```

Notion d'interface (suite)

- Exemple d'utilisation

```
public class Producteur
{
   private final static int MAX_CONSOMMATEURS = 10;
   private Consommateur[] tableConsos;
   private int nbConsos;

   public Producteur()
   {
      this.tableConsos = new Consommateur[MAX_CONSOMMATEURS];
      this.nbConsos = 0;
   }

   public void enregistreConsommateur(Consommateur c)
   {
      if (this.nbConsos<MAX_CONSOMMATEURS)
       this.tableConsos[nbConsos++] = c;
   }

   public void produit (Object p)
   {
      for (int i=0;i<this.nbConsos;i++)
         tableConsos[i].consomme(p);
   }
}
```

Notion d'interface (suite)

```
public class ProducteurDeChaines extends Producteur
{
   public void produitChaine()
   {
     String s = ... (lecture sur l'entrée standard)
     super.produit(s);
   }
}
```

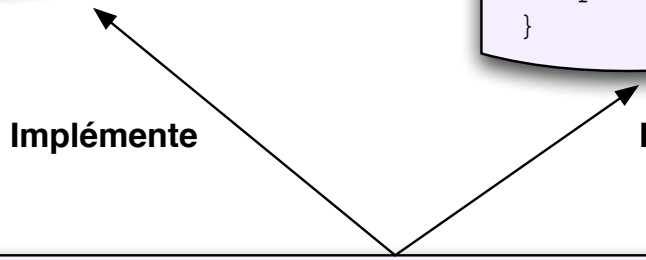
```
public class TestProducteurDeChaines
{
   public static void main(String[]args)
   {
     ProducteurDeChaines pc = new ProducteurDeChaines();
     ConsommateurDeChaines cc = new ConsommateurDeChaines();
     pc.enregistreConsommateur(cc);
     while(true) pc.produitChaine();
   }
}
```

Notion d'interface (suite)

- L'utilisation d'interfaces permet de palier à l'**absence d'héritage multiple**

```
public interface VehiculeMaritime
{
   public void navigue();
}
```

```
public class VehiculeTerrestre
{
   ...

   public void roule() {...}
}
```



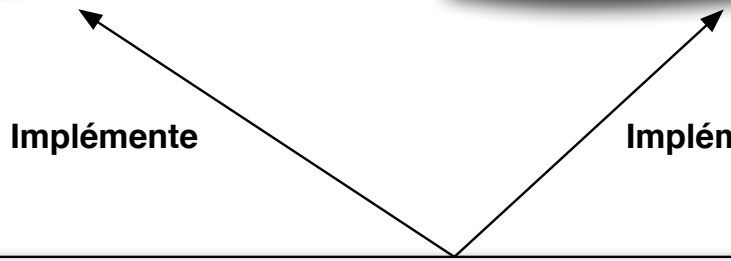

```
public class VehiculeAmphibie extends VehiculeTerrestre
                              implements VehiculeMaritime
{
   ...

   public void navigue() {...}
}
```

Notion d'interface (suite)

- Héritage simple mais **implémentation multiple** de contrats

```
public interface Producteur
{
   public void produit(Object o);
}
```

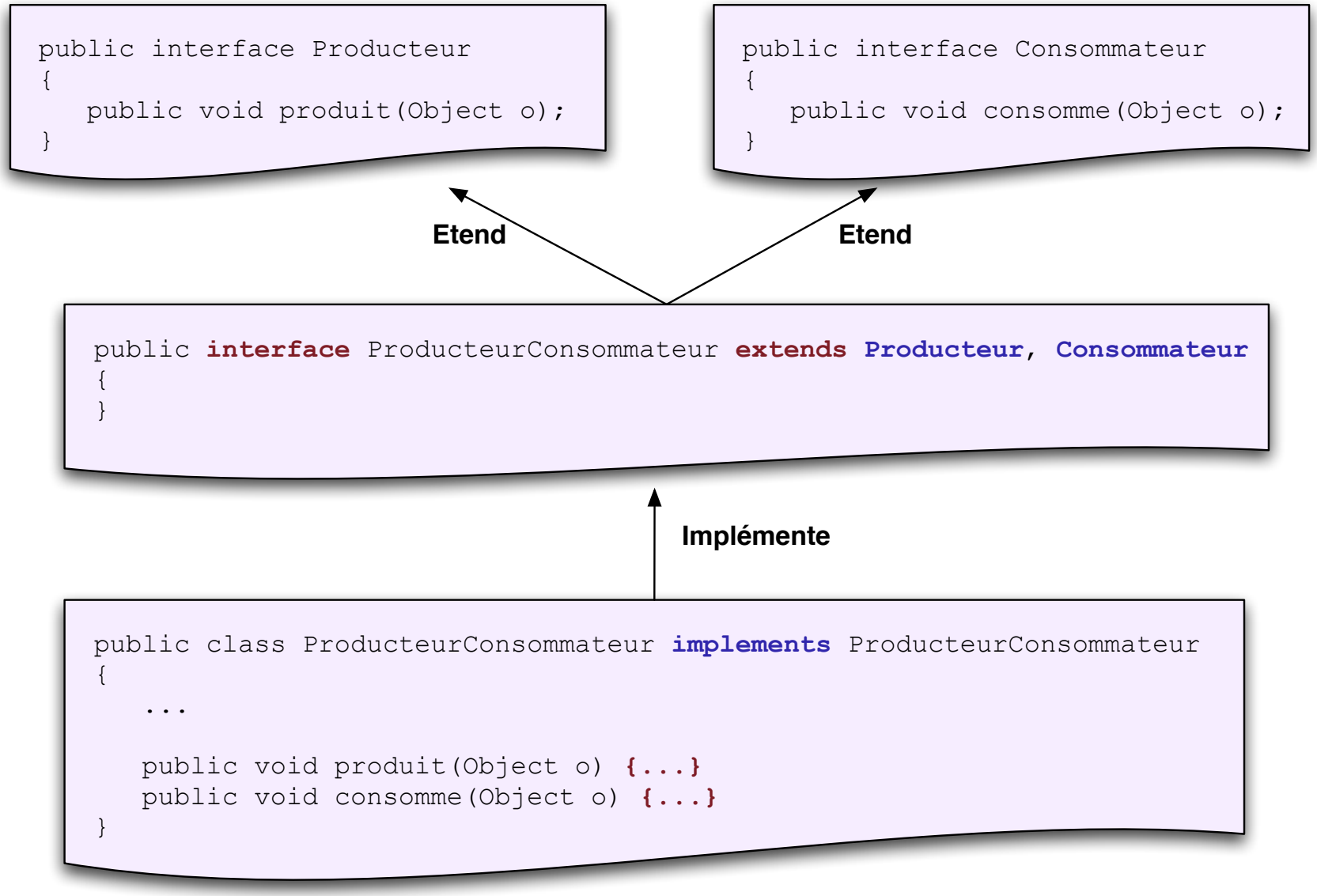
```
public interface Consommateur
{
   public void consomme(Object o);
}
```




```
public class ProducteurConsommateur implements Producteur, Consommateur
{
   ...

   public void produit(Object o) {...}
   public void consomme(Object o) {...}
}
```

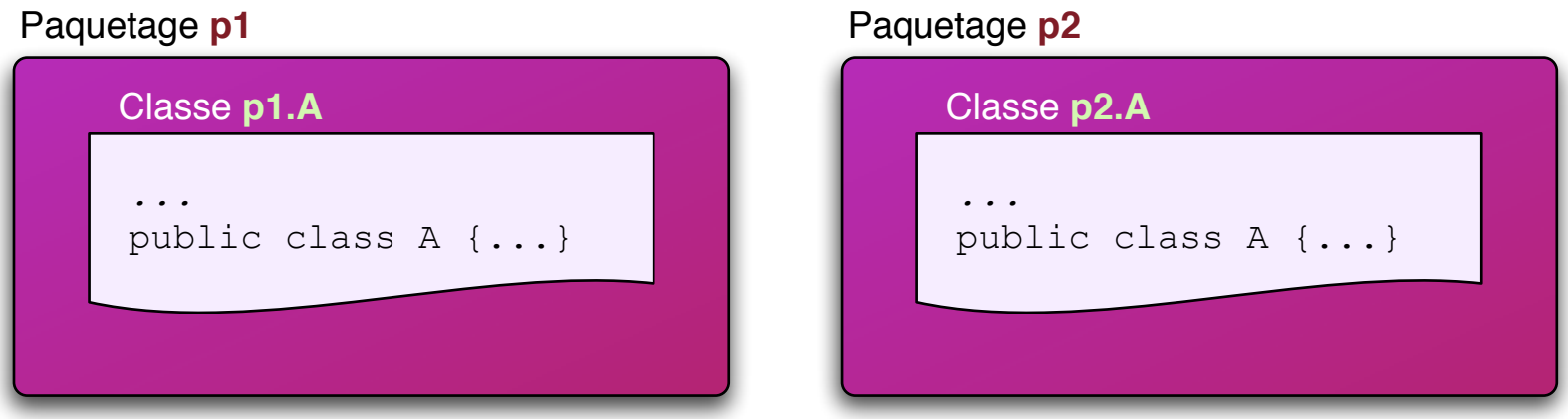
Notion d'interface (suite)

- **Héritage multiple de contrats**



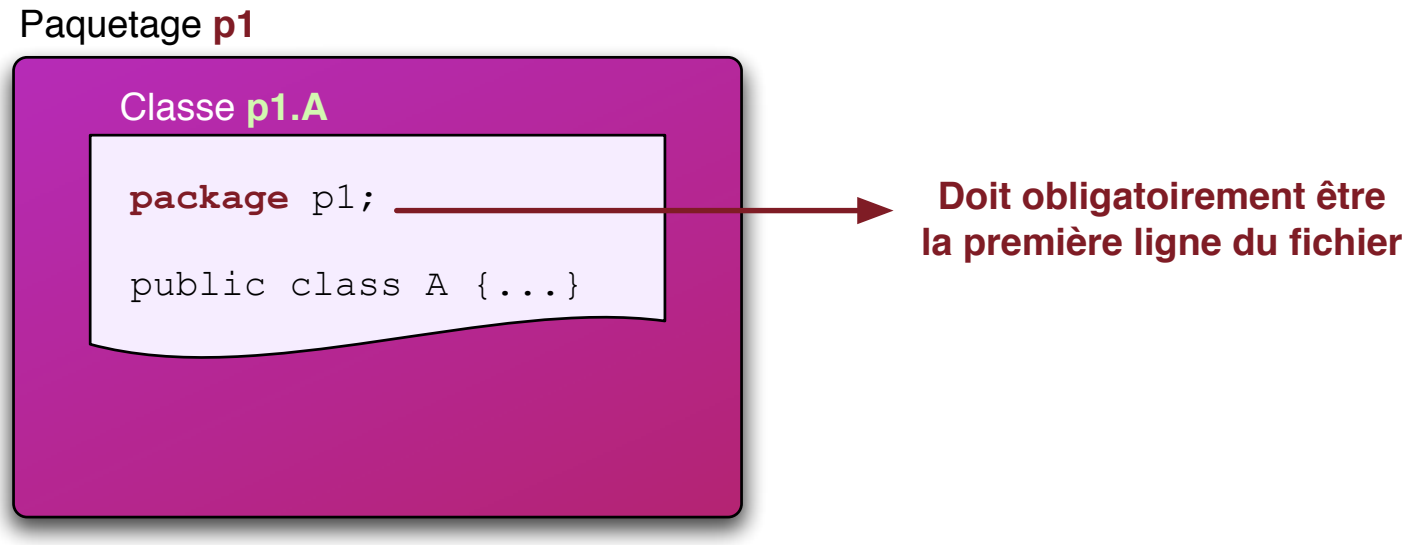
Notion de paquetage

- Un **paquetage** est une **unité logique** renfermant **un ensemble de classes ayant un lien (fonctionnel) entre elles**
- Un paquetage est désigné par un **nom unique** et défini un **espace de nommage** qui permet d'**éviter les conflits de noms**
  - Deux classes peuvent avoir le **même nom « local »** mais elles se distinguent par leur **nom complet**



Notion de paquetage (fin)

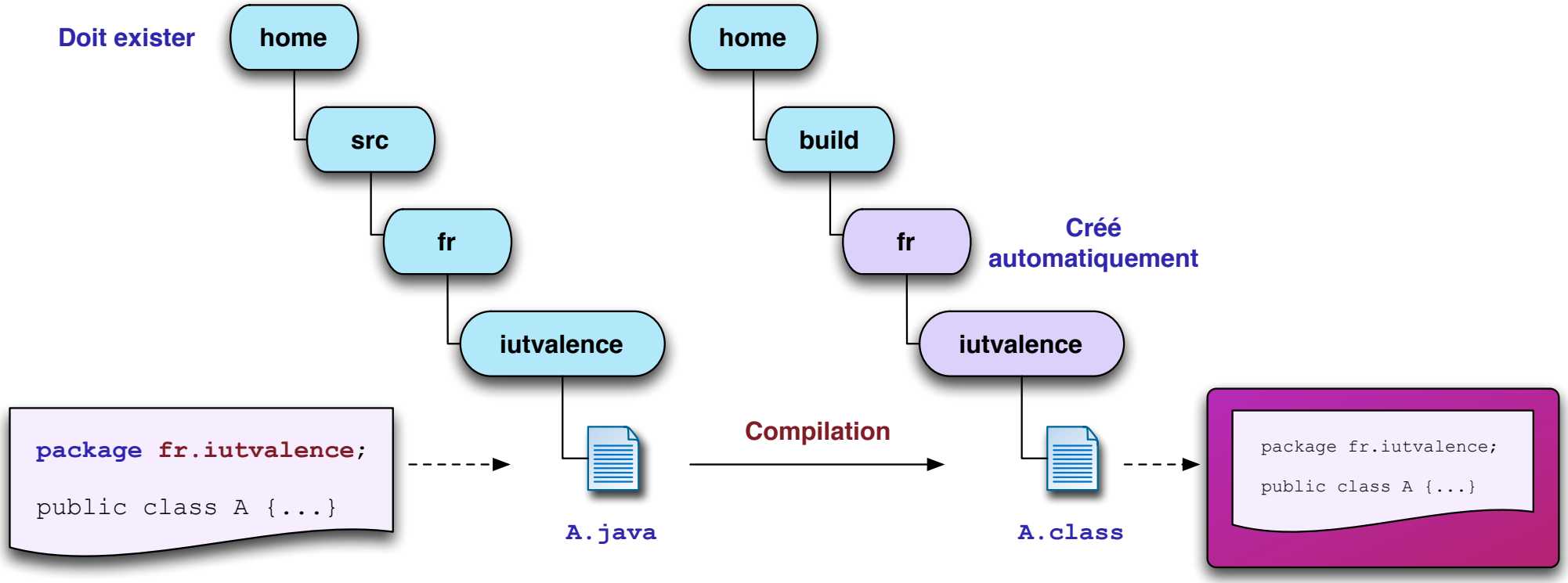
- Un paquetage n'a **pas d'existence physique**
  - Il n'existe **pas de fichier décrivant le contenu du paquetage**
  - Il se construit **virtuellement** par **inclusion de classes**



Compilation d'une classe incluse dans un paquetage

- Ligne de commande (exécutée depuis `/home/src`)

```
javac -d ../build ./fr/iutvalence/A.java
```



Exécution d'une application incluse dans un paquetage

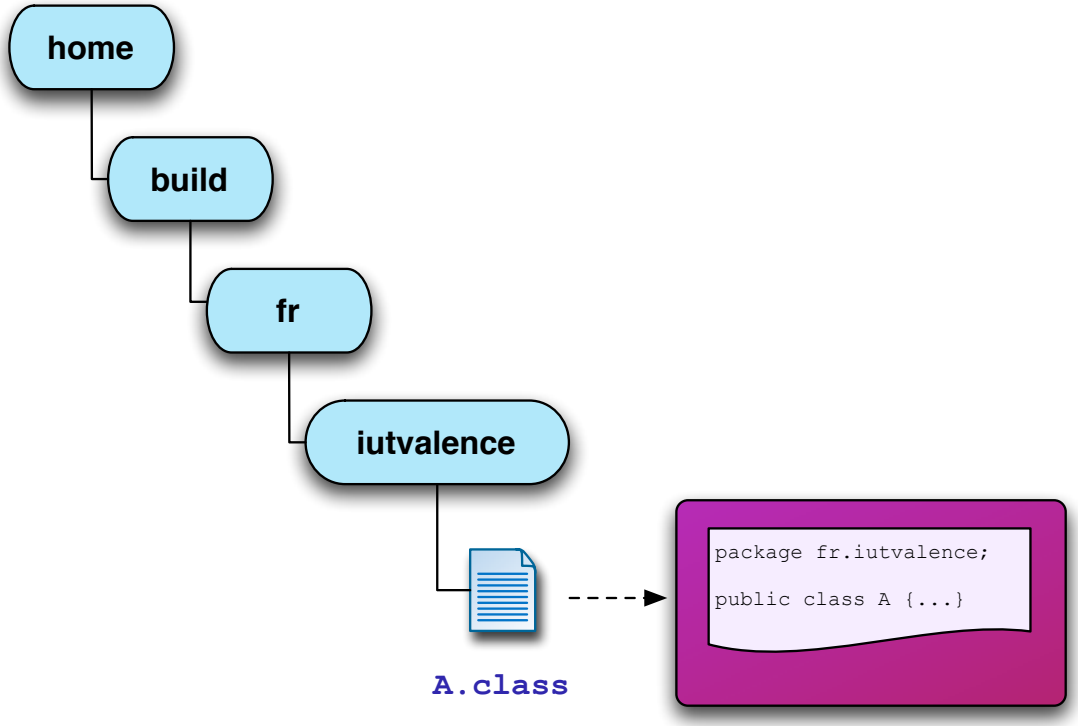
- Ligne de commande (exécutée depuis /home/build)

```
java fr.iutvalence.A
```

Si le répertoire courant n'est pas **build**, il faut nécessairement que le répertoire **build** soit pointé par la variable d'environnement **CLASSPATH**

Si la variable d'environnement **CLASSPATH** n'est pas définie, une **valeur par défaut pointe uniquement sur le répertoire courant**.

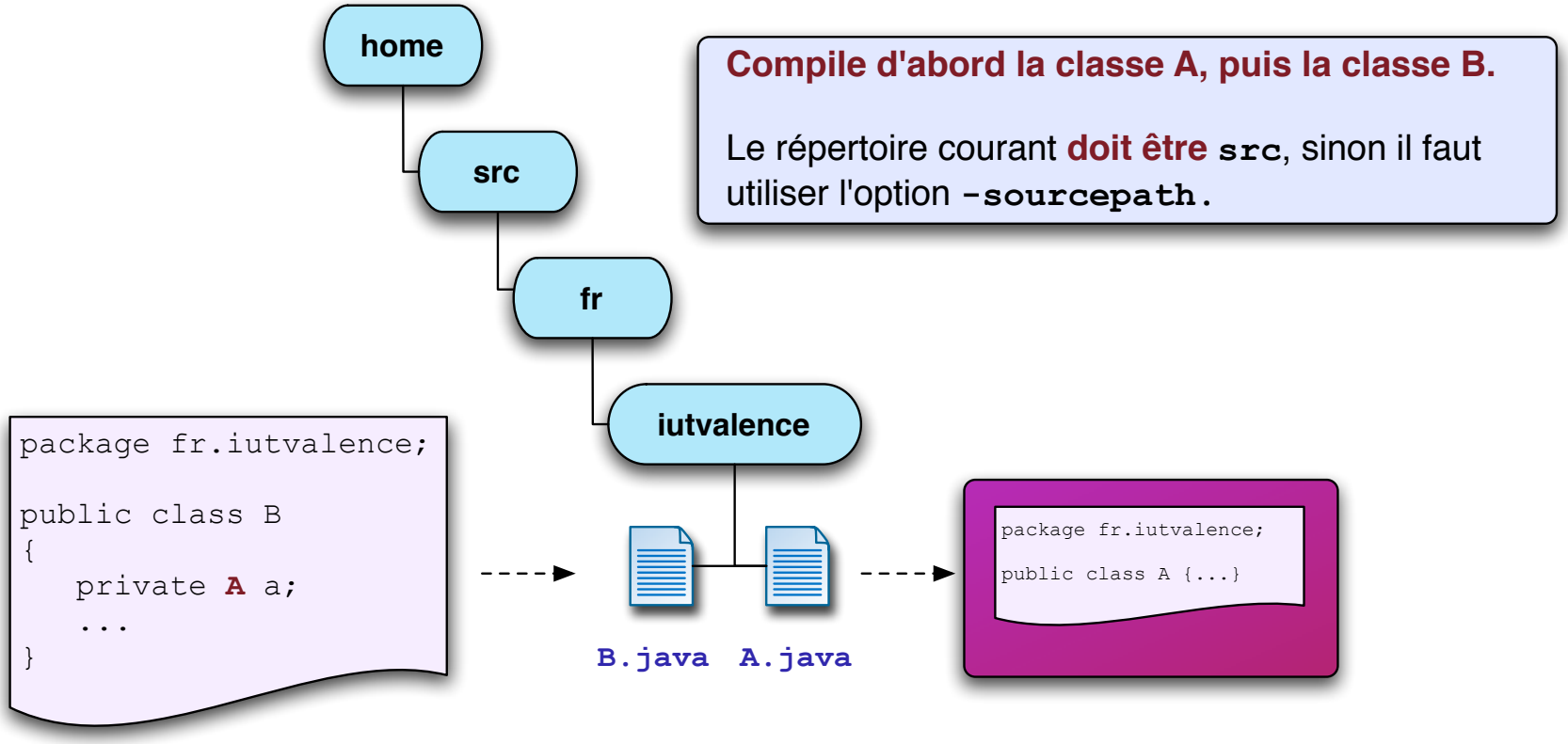
Si la variable est définie, la valeur par défaut n'existe plus.



Edition de lien avec une classe incluse dans le même paquetage

- Cas 1 : la classe `A` n'est pas compilée

```
javac -d ../build ./fr/iutvalence/B.java
```



Edition de lien avec une classe incluse dans le même paquetage

- Cas 2 : la classe `A` est déjà compilée
  - Même ligne de commande que pour simplement compiler une classe incluse dans un paquetage
  - Mais **le répertoire `build` doit être pointé par la variable `CLASSPATH`**

Edition de lien avec une classe incluse dans un autre paquetage

- Il faut spécifier avec le mot-clé `import` (après la directive d'inclusion `package`) le **nom complet de chaque classe utilisée**
  - Dans le code de la classe `B`, on peut utiliser le **nom court des classes importées** s'il n'est **pas ambigü**
  - Si plusieurs classes d'un même paquetage (par ex. `p2`) sont utilisées, il est plus rapide d'écrire `import p2.*;`
    - Remarque : les **hiérarchies de paquetages** n'existent pas, `import fr.*;` n'implique pas `import fr.iutvalence.*;`
- Il faut qu'**un des répertoires pointés par la variable** `CLASSPATH` soit **racine de l'arborescence du paquetage** `fr.iutvalence`

Retour sur la variable CLASSPATH

- La **variable d'environnement** `CLASSPATH` contient une **liste de chemins** conduisant à
  - Des **répertoires** (contenant des **classes** ou des **arborescences de paquetages**)
  - Des **archives *jar***
- Lors de la **compilation** et de l'**exécution** cette variable sert à **localiser les fichiers** `.class` **nécessaires à l'édition de lien**

Paquetages Java standards

- Liste non exhaustive (plus de 100 paquetages au total)
  - `java.lang` (importé implicitement)
    - Classes fondamentales (`Object`, `Class`, ...)
  - `java.util`
    - Classes utilitaires (gestion de collections d'objets, ...)
  - `java.io`
    - Classes pour la gestion des entrées/sorties
  - `java.awt`
    - Classes pour la gestion des interfaces graphiques
  - `java.net`
    - Classes pour la conception d'applications distribuées TCP/UDP

Java et la gestion des erreurs

- L'exécution d'un programme **quel qu'il soit peut engendrer des erreurs**
- Il existe différentes approches du traitement d'erreur
  - **Mélanger** la logique de l'application avec la gestion d'erreur
    - Approche utilisée en *C*, où les valeur de retour des fonctions sont souvent également utilisées pour signaler des erreurs
  - **Séparer** la logique de l'application de la gestion d'erreur
    - Approche utilisée en ***Java***, avec les **exceptions**
    - Il devient possible de **traiter proprement un erreur non pas où elle survient, mais plus tard** (i.e. dans le code appelant)

Notion d'exception

- Une **exception** est un **objet Java** dont la classe hérite de `java.lang.Throwable`, représentant un **type d'erreur**
  - Dépassement de tableau, affectation de valeur invlaide, ...
- Lorsqu'une **erreur survient**, le code l'ayant détecté **soulève explicitement une nouvelle exception du type adéquat**
  - Création d'un nouvel objet + signalement explicite
  - L'**exécution de la méthode s'arrête** et le flot de contrôle retourne à l'appelant (avec l'exception)

Notion d'exception (suite)

- Une méthode **soulevant une ou plusieurs exceptions** doit le **signaler à l'appelant**
  - **Ajout d'information dans la signature de la méthode**
- Une méthode **appelant une méthode signalant une exception doit soit**
  - **Propager explicitement** cette exception
    - L'**exécution de la méthode s'arrête** et le flot de contrôle revient à l'appelant (avec l'exception)
  - **Capturer explicitement et traiter** cette exception
    - L'**exécution de la méthode se poursuit normalement**

Types d'exceptions

- Exceptions **contrôlées**
  - Héritent de `java.lang.Exception` (sous-classe de `Throwable`
  - **Le compilateur vérifie** qu'une **méthode susceptible de soulever une exception contrôlée** soit **le signale à l'appelant**, soit **traite l'exception elle-même**
- Exceptions **non contrôlées**
  - Héritent de `java.lang.Error` (sous-classe de `Throwable`) ou de `java.lang.RuntimeException` (sous-classe d'`Exception`)
  - **Pas de vérification du compilateur** ni d'**obligation pour les méthodes de signaler ou de traiter l'exception**

Signaler une exception

- Utilisation du mot-clé `throws`

```
package java.io;

public abstract class Inputstream
{
   ...

   public abstract int read()throws IOException;
 }
```

Propager une exception

- Simplement un **re-signalement**

```
import java.io.*;

public class Test
{
    ...

    public int plop(FileInputStream f) throws IOException
    {
        int lu = f.read();
        return lu;
    }
 }
```

Capturer et traiter une exception

- Utilisation de la clause `try ...catch ...finally`

```
public class Test
{
   ...

   public int plop(FileInputStream f)
   {
      int lu = 0;

      try                                   On tente l'exécution normale
      {
         lu = f.read();
         ... // code pouvant soulever d'autres exceptions
      }

      catch (IOException e)   On capture les exceptions soulevées
      {
         ... // Traitement de l'exception
      }
      catch (...) {...}          (en différenciant le traitement
                                  en fonction de l'erreur)

      finally
      {
         ... // Code exécuté dans tous les cas
      }
      return lu;
   }
}
```

La classe Exception

Méthodes de la classe java.lang.Exception		
public	Exception()	*Constructeur*
public	Exception(String message)	*Constructeur.* `message` *permet de décrire l'exception en langage naturel*
public	Exception(String message, Throwable cause)	*Constructeur. Idem précédent,* `cause` *permet de chaîner avec une exception précédente dans le cas d'une capture/re-soulèvement*
public	Exception(Throwable cause)	*Constructeur. Idem précédent sans message*
public Throwable	getCause()	*Obtention de l'exception originale sans le cas d'une capture/re-soulèvement*
public String	getMessage()	*Obtention du message associé à l'exception*
public void	printStackTrace()	*Affichage de la pile d'exception sur la sortie standard*
public void	printStackTrace(PrintStream ps)	*Ecriture de la pile d'exception sur un flux de sortie texte*

Soulever une exception

- Utilisation du mot-clé `throw`

```
public class EntierPositif
{
   private int value;

   public EntierPositif() {this.value = 0;}

   public EntierPositif (int value) throws BadValueException {this.setValue(value);}

   public int getValue() {return this.value;}

   public void setValue(int value) throws BadValueException
   {
      if (value >= 0) this.value = value;
      else throw new BadValueException();
   }
}
```

java.io.File

- L'API `java.io` permet entre autres **la manipulation de fichiers et répertoires** à travers la classe `File`
- Cette classe permet la **représentation sous forme objets de chemins de désignation de fichiers ou de répertoires dans des systèmes de fichiers locaux ou réseaux**
  - Il n'est pas obligatoire que les **fichiers ou répertoires désignés existent**
  - Les chemins peuvent être **absolus** ou **relatifs**
  - Les chemins sont **représentés en interne de manière indépendante des conventions du système de fichiers** (séparateurs, ...)
- A travers la désignation de fichiers et de répertoires, la classe `File` offre des **opérations de manipulation élémentaires**
  - **Création, suppression, renommage, ...**

java.io.File (suite)

Méthodes de la classe java.io.File		
public static String	pathSeparator	*Attribut représentant le séparateur de chemins pour construire des listes de chemins sur la plate-forme courante, exprimé sous la forme d'une chaîne de caractères.*
public static char	pathSeparatorChar	*Idem précédent, mais le séparateur est exprimé sous la forme d'un caractère.*
public static String	separator	*Attribut représentant le séparateur de répertoires pour construire des chemins sur la plate-forme courante, exprimé sous la forme d'une chaîne de caractères.*
public static char	separatorChar	*Idem précédent, mais le séparateur est exprimé sous la forme d'un caractère.*
public	File(String path)	*Création d'un nouvel objet `File` représentant le fichier/répertoire désigné par **path**.*
public	File(String parent, String path)	*Création d'un nouvel objet `File` représentant le fichier/répertoire désigné par **parent+separator+child**.*
public	File(File parent, String path)	*Idem précédente, mais le répertoire parent est exprimé sous la forme d'un objet `File`.*

java.io.File (suite)

- Manipulation de chemins

Méthodes de la classe java.io.File		
public String	getPath()	*Obtention du chemin complet sous la forme d'une chaîne de caractères (en utilisant le séparateur de répertoires courant).*
public String	getName()	*Obtention du nom du fichier ou du répertoire (seulement le dernier élément du chemin) sous la forme d'une chaîne de caractères (en utilisant le séparateur de répertoires courant).*
public boolean	isAbsolute()	*Test du caractère absolu de la représentation du chemin.*
public String	getAbsolutePath()	*Obtention d'une forme absolue équivalente au chemin désigné, sous la forme d'une chaîne de caractères (en utilisant le séparateur de répertoires courant).*
public File	getAbsoluteFile()	*Idem précédente, mais sous la forme d'un objet `File`.*
public String	getParent()	*Obtention du répertoire parent du chemin désigné, sous la forme d'une chaîne de caractères (en utilisant le séparateur de répertoires courant).*
public File	getParentFile()	*Idem précédente, mais sous la forme d'un objet `File`.*

java.io.File (suite)

- Obtention d'informations sur les fichiers/répertoires

Méthodes de la classe java.io.File		
public static File[]	listRoots()	*Obtention de la liste des racines du système de fichiers.*
public String[]	list()	*Obtention de la liste des noms des fichiers et sous-répertoires contenu dans le répertoire désigné par l'objet `File`. Retourne `null` si l'objet désigne un fichier.*
public File[]	listFiles()	*Idem précédente, mais retourne un tableau d'objets `File`.*
public String[]	list(FileNameFilter f)	*Obtention de la liste des noms des fichiers et sous-répertoires du répertoire désigné par l'objet `File` et respectant un modèle de syntaxe exprimé par le filtre `f`.*
public File[]	listFiles(FileNameFilter f)	*Idem précédente, mais retourne un tableau d'objets `File`.*

Méthodes de l'interface java.io.FilenameFilter		
public boolean	accept(File parent, String name)	*Test du passage du filtre pour le fichier/répertoire de nom `name` du répertoire `parent`.*

java.io.File (suite)

- Obtention d'informations sur les fichiers/répertoires (suite)

Méthodes de la classe `java.io.File`		
public boolean	exists()	*Test d'existence.*
public boolean	isDirectory()	*Test d'existence du répertoire désigné par l'objet `File`.*
public boolean	isFile()	*Idem précédente, mais pour un fichier.*
public boolean	isHidden()	*Test de masquage par le système de fichiers.*
public boolean	canRead()	*Test de droit en lecture.*
public boolean	canWrite()	*Test de droit en écriture.*
public long	length()	*Obtention de la taille du fichier. Retourne 0 si le le fichier n'existe pas où si l'objet désigne un répertoire.*
public long	lastModified()	*Obtention de la date de dernière modification du fichier, exprimé sous la forme du nombre de millisecondes écoulées depuis le 01/01/1970 à 00 :00 :00 GMT.*

java.io.File (suite)

- Manipulation de fichiers/répertoires

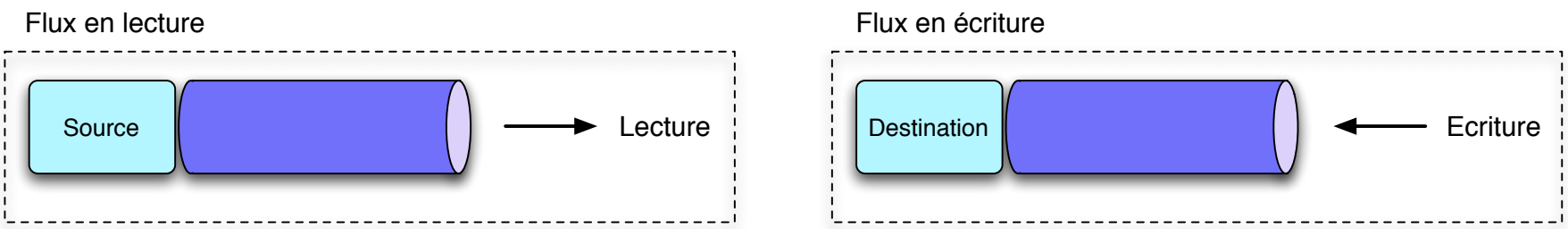
Méthodes de la classe java.io.File		
public boolean	createNewFile()	*Création d'un nouveau fichier à l'emplacement désigné par l'objet `File`. Retouren `true` si le fichier à été créé, `flase` si le fichier existait déjà. Soulève `IOException` si le fichier n'a pas pu être créé.*
public static boolean	createTempFile(String p, String e, File dir)	*Création d'un fichier temporaire (pas obligatoirement effacé à la fin de l'application) dans le répertoire `dir`, dont le nom généré automatiquement porte l'extension `e` et débute par le préfixe `p` (au moins 3 caractères) Soulève `IOException` et `IllegalArgumentException`.*
public static boolean	createTempFile(String prefix, String ext)	*Idem précédente, mais création dans le répertoire temporaire par défaut.*
public boolean	delete()	*Suppression d'un fichier ou d'un répertoire vide.*
public boolean	deleteOnExit()	*Marquage pour suppression à la terminaison de l'application.*
public boolean	mkdir()	*Création d'un répertoire (le répertoire parent doit obligatoirement exister.*
public boolean	mkdirs()	*Idem précédente mais le crée l'arborescence manquante si besoin.*

java.io.File (fin)

- Manipulation de fichiers/répertoires

Méthodes de la classe java.io.File		
public boolean	renameTo(File f)	*Renommage d'un fichier ou d'un répertoire.*
public boolean	setLastModified(long ms)	*Modification de la date de dernière modification du fichier. La date est exprimée en nombre de millisecondes écoulées depuis la date référence.*
public boolean	setReadOnly()	*Positionnement des droits en lecture seule.*

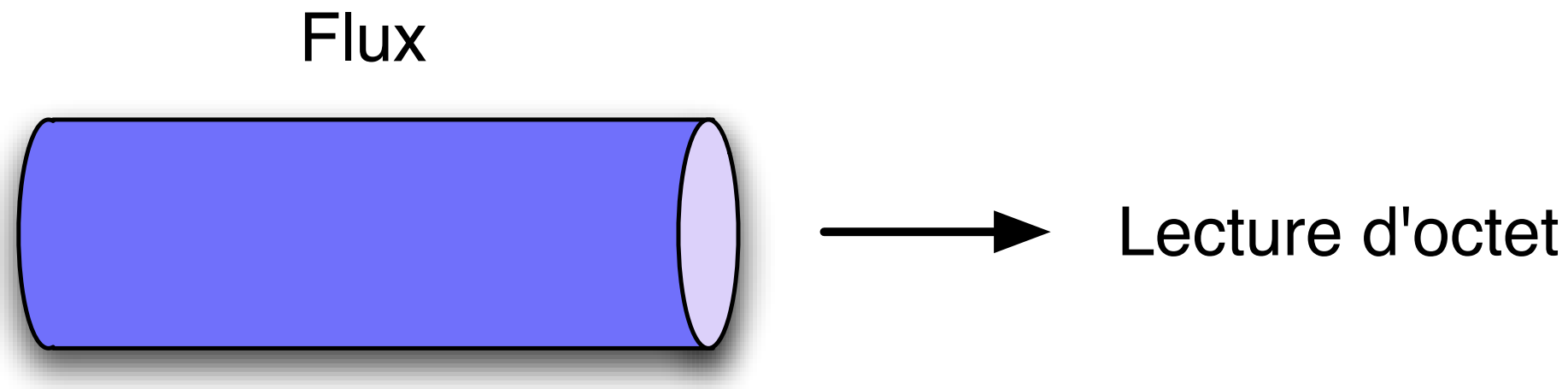
Notion de flux

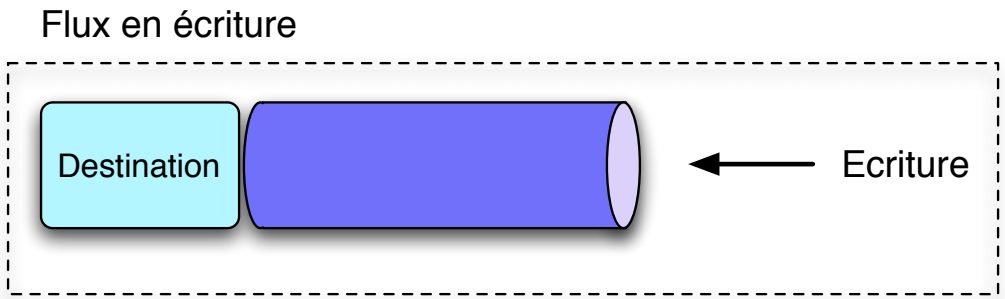


- La gestion des entrées-sorties en Java s'appuie sur la notion de **flux** (***stream***)
  - **Séquence ordonnée d'informations** possédant une **source** ou une **destination**
- Dans un flux, des données sont **produites** (écrites) et **consommées** (lues) **de manière séquentielle**
  - Les données sont lues **dans l'ordre** où elles ont été écrites
  - Le **rythme d'écriture** peut être **différent** du **rythme de lecture**
  - Le flux peut comporter un **tampon**

Notion de flux (fin)

- Utilisation d'un flux

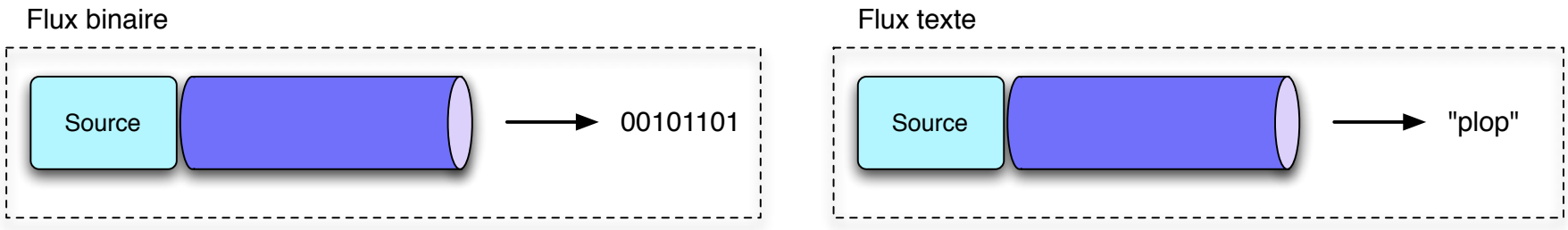





```
Ouvrir le flux
Tant qu'il y a de l'information à écrire
   écrire de l'information
Fin Tant que
Fermer le flux
```

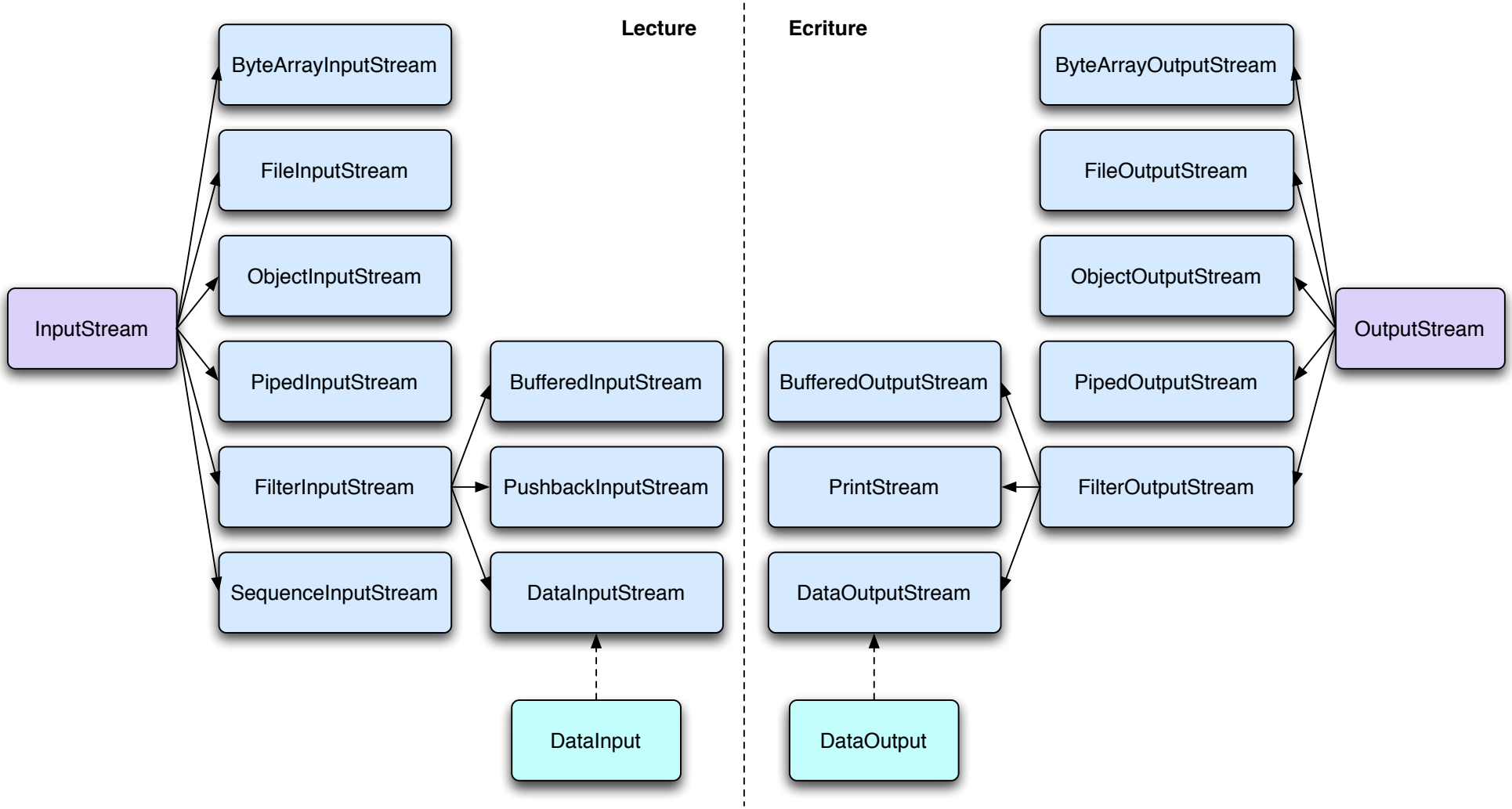
Introduction au paquetage java.io

- `java.io` regroupe un ensemble d'interfaces et de classes pour la **lecture/écriture synchrone dans des flux** divers
- Deux hiérarchies disctinctes

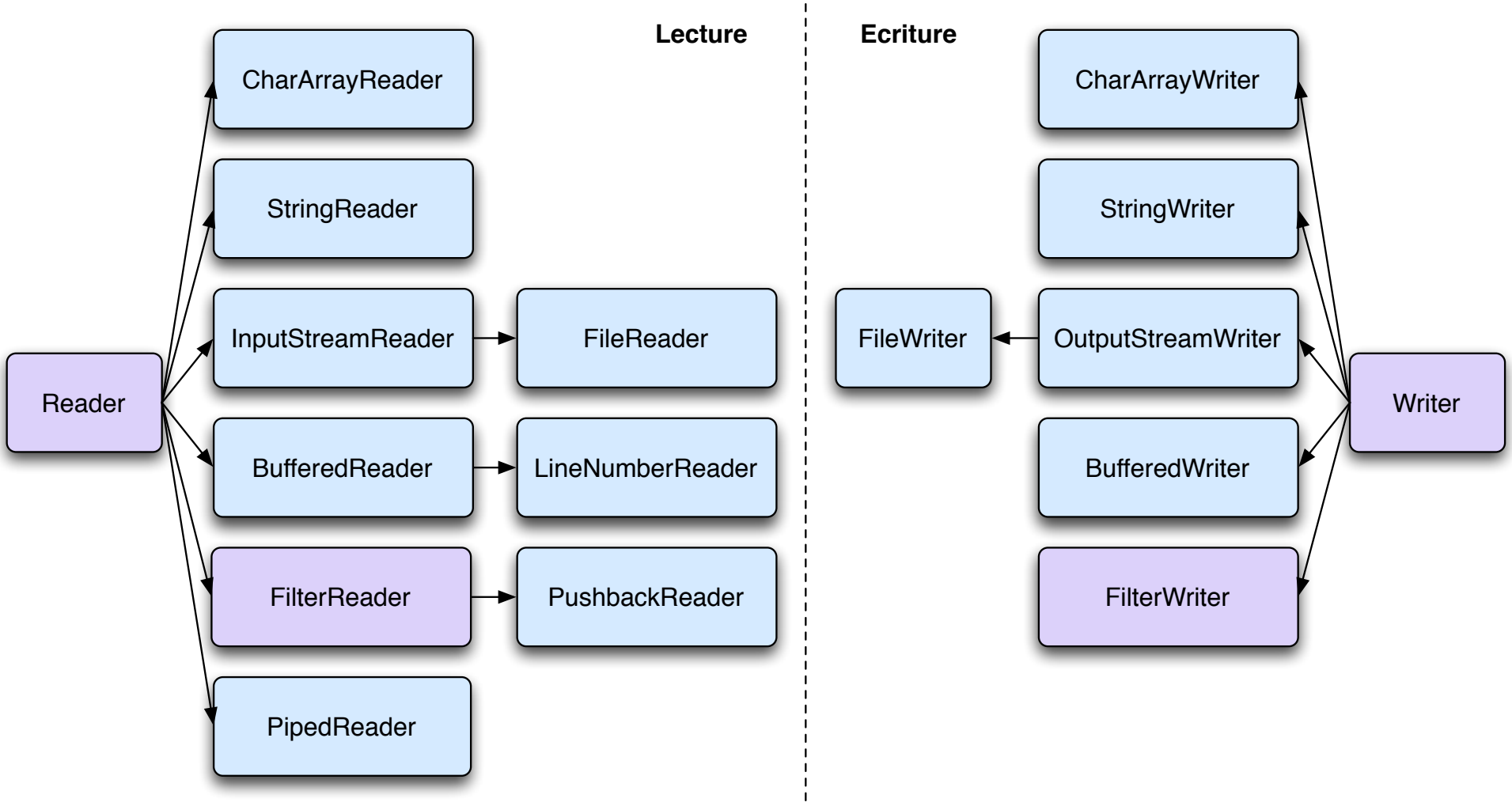


  - `InputStream, OutputStream`
    - Lecture/écriture d'**octets**
    - Flux binaire brut
  - `Reader, Writer`
    - Lecture/écriture de **caractères**
    - Flux texte, notion d'**encodage des caractères**

Introduction au paquetage java.io (suite)

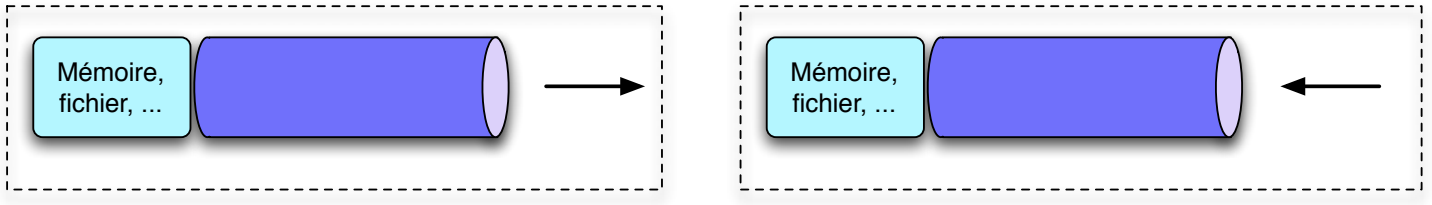


Introduction au paquetage java.io (fin)



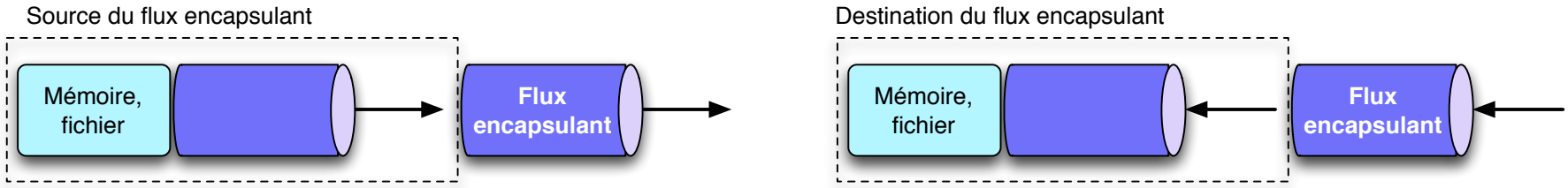
Types de flux

- **Flux « primitifs »**



  - Fournissent **directement** des **abstractions de base**
  - **Lire/écrire des octets en mémoire**, à partir de/dans un **fichier** ou un **tube de communication**, . . .

- **Flux « encapsulants »**



  - Fournissent des **abstractions de plus haut niveau en s'appuyant sur des flots primitifs**
  - **Lire/écrire des octets avec gestion de tampon**, . . .

java.io.InputStream

Méthodes de la classe abstraite java.io.InputStream		
public int	available()	*Obtention du nombre d'octets disponibles à la lecture. Soulève `IOException` en cas de problème.*
public void	close()	*Fermeture du flux. Soulève `IOException` en cas de problème.*
public abstract int	read()	*Lecture **bloquante** d'un octet de données. Retourne l'octet lu sous la forme d'un entier dont la valeur est comprise entre 0 et 255 ou -1 si la fin du flux a été atteinte. Soulève `IOException` en cas de problème.*
public int	read(byte[] b)	*Lecture **bloquante** d'un ensemble d'octets, stockés dans le tableau `b`. Retourne le nombre d'octets lus. Soulève `IOException` en cas de problème.*
public int	read(byte[] b, int off, int l)	*Idem précédente mais lecture d'au plus `l` octets stockés à partir de l'indice `off`.*
public long	skip(long b)	*Non-lecture (sortis du flux mais ignorés) d'au plus `b` octets. Retourne le nombre d'octets ignorés. Soulève `IOException` en cas de problème.*
public boolean	markSupported()	*Test de la capacité du flux à mémoriser une position. **Optionnel**.*
public void	mark(int n)	*Mémorisation de la position courante, invalidée après la lecture de `n` octets. **Optionnel**.*
public void	reset()	*Retour à la dernière position mémorisée. Soulève `IOException` en cas de problème. **Optionnel***

java.io.OutputStream

Méthodes de la classe **abstraite** java.io.OutputStream		
public void	close()	*Fermeture du flux. Soulève `IOException` en cas de problème.*
public abstract void	write(byte b)	*Ecriture d'un octet dans le flux. Soulève `IOException` en cas de problème.*
public void	write(byte[] b)	*Ecriture d'un ensemble d'octets, stockés dans le tableau `b`. Soulève `IOException` en cas de problème.*
public void	write(byte[] b, int off, int l)	*Idem précédente mais écriture d'au plus `l` octets stockés à partir de l'indice `off`.*
public void	flush()	*Ecriture forcée des octets mis en tampon (dans le cas d'un flux avec gestion de tampon). Soulève `IOException` en cas de problème.* **Optionnel**.

Flux primitifs

- Lire et écrire des octets **en mémoire** (i.e. dans/vers un tableau d'octets)
  - `ByteArrayInputStream`, `ByteArrayOutputStream`
- Lire et écrire des octets dans/vers un **fichier binaire**
  - `FileInputStream`, `FileOutputStream`
- Lire et écrire des octets dans/vers un **tube de communication**
  - `PipedInputStream`, `PipedOutputStream`
  - Nécessité d'**associer explicitement** les flux de lecture et d'écriture (via les constructeurs) pour **créer le tube**

FileInputStream / FileOutputStream

Constructeurs de la classe java.io.FileInputStream		
public	FileInputStream(String path)	*Création d'un flux permettant de lire des octets dans le fichier désigné par le chemin `path`. Soulève `FileNotFoundException` si le fichier n'existe pas.*
public	FileInputStream(File path)	*Idem précédente mais le chemin est passé sous la forme d'un objet `File`.*

Constructeurs de la classe java.io.FileOutputStream		
public	FileOutputStream(String path)	*Création d'un flux permettant d'écrire dans le fichier désigné par le chemin `path`. Si le fichier existe, il est écrasé. Si le fichier n'existe pas, il est créé. Soulève `FileNotFoundException` si le fichier n'existe pas et ne peut être créé.*
public	FileOutputStream(File path)	*Idem précédente mais le chemin est passé sous la forme d'un objet `File`.*
public	FileOutputStream(String path, boolean append)	*Idem `FileOutputStream(String)` mais il est possible de spécifier si, dans le cas d'un fichier existant, il y a écrasement des données existantes ou ajout en fin de fichier.*
public	FileOutputStream(File path, boolean append)	*Idem précédente mais le chemin est passé sous la forme d'un objet `File`.*

FileInputStream / FileOutputStream (suite)

- Exemple : copie binaire de fichiers

```
public static void copyFile(String srcPath, String destPath)
{
   File srcFile = new File(srcPath);
   FileInputStream fis = null;
   FileOutputStream fos = null;
   try
   {
      fis = new FileInputStream(srcFile);
      fos = new FileOutputStream(destPath);
   }
   catch (FileNotFoundException e) {...}
   long resteALire = srcFile.length();
   try
   {
      while (resteALire>0)
      {
         int av = fis.available();
         byte[] buf= new byte[av];
         fis.read(buf,0,av);
         fos.write(buf,0,av);
         resteALire -= av;
      }
      fos.close();
      fis.close();
   }
   catch (IOException e) {...}
}
```

ByteArrayInputStream / ByteArrayOutputStream

Constructeurs de la classe java.io.ByteArrayInputStream		
public	ByteArrayInputStream(byte[] b)	*Création d'un flux permettant de lire des octets dans le tableau `b`.*
public	ByteArrayInputStream(byte[] b, int off, int l)	*Idem précédente mais il y a au plus `l` octets à lire et ces octets sont stockés à partir de l'indice `off`.*

Méthodes de la classe java.io.ByteArrayOutputStream		
public	ByteArrayOutputStream()	*Création d'un flux permettant d'écrire dans un tableau d'octets interne.*
public	ByteArrayOutputStream(int cap)	*Idem précédente mais le nombre d'octets pouvant être écrits est limité à `cap`.*
public byte[]	toByteArray()	*Obtention du tableau interne contenant les octets écrits.*
public void	writeTo(OutputStream o)	*Recopie (de tous les octets écrits) vers le flux binaire `o`. Soulève `IOException` en cas de problème d'écriture sur le flux de sortie.*

ByteArrayInputStream / ByteArrayOutputStream (suite)

- Exemple : lecture en mémoire de fichiers binaires

```
public static byte[] getFileBytes(String srcPath)
{
   byte[] result = null;
   File srcFile = new File(srcPath);
   FileInputStream fis = null;
   ByteArrayOutputStream bos = null;
   try {fis = new FileInputStream(srcFile);}
   catch (FileNotFoundException e) {...}
   bos = new ByteArrayOutputStream();
   long resteALire = srcFile.length();
   try
   {
      while (resteALire>0)
      {
         int av = fis.available();
         byte[] buf= new byte[av];
         fis.read(buf,0,av);
         bos.write(buf,0,av);
         resteALire -= av;
      }
      result = bos.toByteArray();
      bos.close();
      fis.close();
   }
   catch (IOException e) {...}
   return result;
}
```

Flux encapsulants

- Lire des octets **dans une suite de flux** (i.e. lire tous les octets du premier, puis du second, ...)
  - `SequenceInputStream`
- Lire et écrire des octets **à travers un tampon** (flux à mémoire)
  - `BufferedInputStream`, `BufferedOutputStream`
- Lire et écrire des octets avec **possibilité de retour arrière**
  - `PushbackInputStream`
- Lire et écrire des **valeurs de types primitifs**
  - `DataInputStream`, `DataOutputStream`
- Ecrire des **représentations textuelles de valeurs de types primitifs**
  - `PrintStream`

Interfaces DataInput / DataOutput

Méthodes de l'interface java.io.DataInput		
public boolean	readBoolean()	*Lecture d'une valeur booléenne. Soulève* `IOException` *si la lecture échoue.*
public byte	readByte()	*Lecture d'un octet. Soulève* `IOException` *si la lecture échoue.*
public char	readChar()	*Lecture d'un caractère (UTF-16). Soulève* `IOException` *si la lecture échoue.*
public short	readShort()	*Lecture d'un entier court. Soulève* `IOException` *si la lecture échoue.*
public int	readInt()	*Lecture d'un entier. Soulève* `IOException` *si la lecture échoue.*
public long	readLong()	*Lecture d'un entier long. Soulève* `IOException` *si la lecture échoue.*
public float	readFloat()	*Lecture d'un flottant simple précision. Soulève* `IOException` *si la lecture échoue.*
public double	readDouble()	*Lecture d'un flottant double précision. Soulève* `IOException` *si la lecture échoue.*

Interfaces DataInput / DataOutput (fin)

Méthodes de l'interface java.io.DataInput		
public void	readFully(byte[] b)	*Lecture bloquante tout-ou-rien d'un nombre exact d'octets (égal à la capacité du tableau `b`, où seront stockés les octets lus). Soulève `IOException` si la lecture échoue, ou `EOFException` si la fin du flux est atteinte avant d'avoir pu lire tous les octets.*
public void	readFully(byte[] b, int off, int l)	*Idem précédente, mais stockage dans un sous-tableau.*
public int	skipBytes(int l)	*Lecture, puis oubli, d'au plus `l` octets. Retourne le nombre d'octets lus. Soulève `IOException` si la lecture échoue.*

- L'interface `DataOutput` propose des méthodes similaires pour écrire des valeurs de types primitifs
  - `writeBoolean(boolean)`, `writeChar(char)`, ...

DataInputStream / DataOutputStream

- DataInputStream
  - Etend FilterInputStream, implémente DataInput

Constructeurs de la classe java.io.DataInputStream		
public	DataInputStream(InputStream is)	*Création d'un flux permettant de lire des valeurs de types primitifs via le flux de lecture d'octets is.*

- DataOutputStream
  - Etend FilterOutputStream, implémente DataOutput

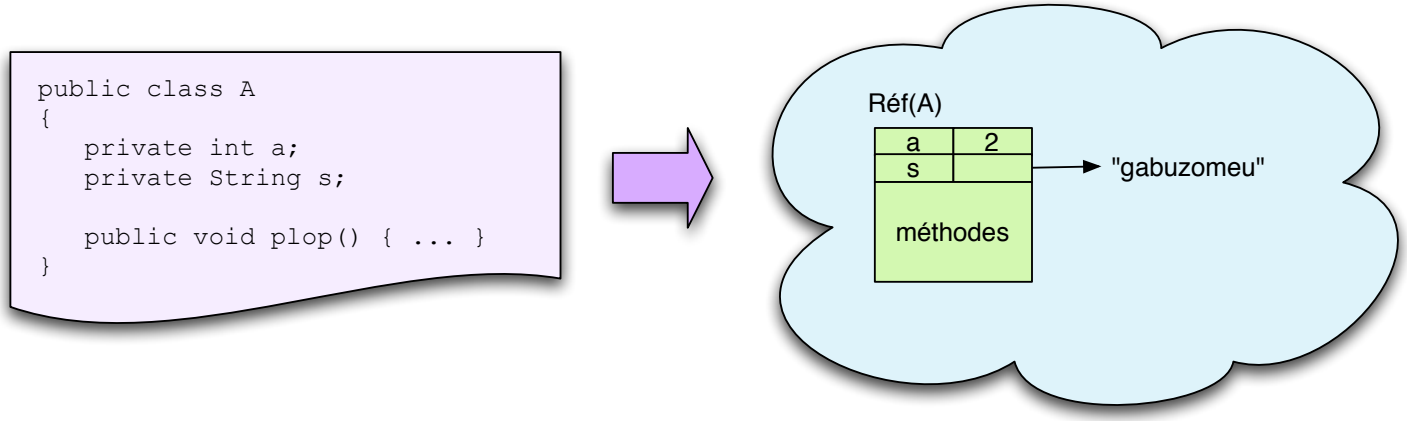
Méthodes de la classe java.io.DataOutputStream		
public	DataOutputStream(OutputStream os)	*Création d'un flux permettant d'écrire des valeurs de types primitifs via le flux d'écriture d'octets os.*
public int	size()	*Obtention du nombre d'octets écrits dans le flux.*

DataInputStream / DataOutputStream (fin)

- Exemple : lecture d'entiers dans un fichier binaire

```
public static int[] getFileInts(String srcPath)
{
   File srcFile = new File(srcPath);
   DataInputStream dis = null;
   try
   {
      dis = new DataInputStream(new FileInputStream(srcFile));
   }
   catch (FileNotFoundException e) {...}
   int[] result = new int[srcFile.length()/4];
   try
   {
      for (int i = 0; i<result.length; i++)
         result[i] = dis.readInt();
      dis.close();
      fis.close();
   }
   catch (IOException e) {...}
   return result;
}
```

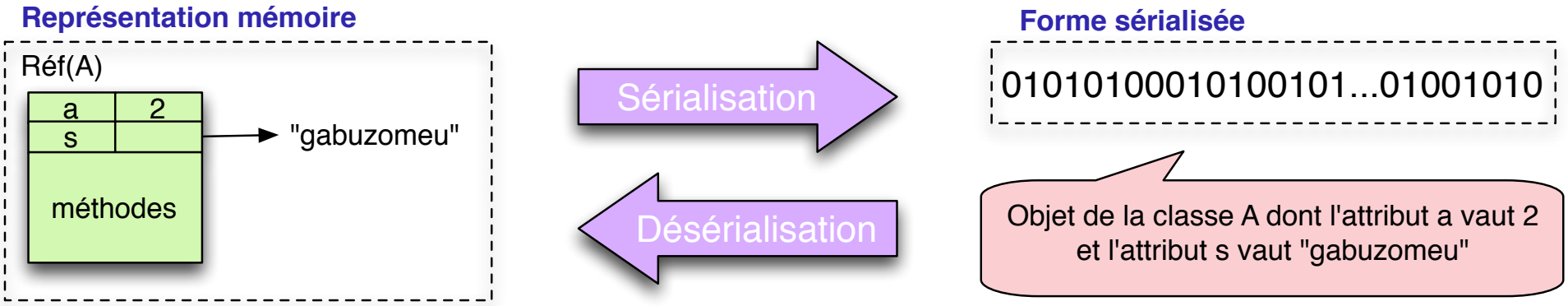
Sérialisation d'objets Java



- Au cours de l'exécution d'une application Java, les objets sont **en mémoire**
  - **Si la machine virtuelle s'arrête** (normalement ou anormalement), **tous les objets instanciés depuis le démarrage sont ramassés**
- Dans certains cas, il est souhaitable de pouvoir **stocker ces objets sur un support persistant**
  - Pour être **tolérant aux fautes**, pour **échanger des objets entre deux JVM**, ...

Sérialisation d'objets Java (suite)

- Problème : la **représentation memoire** n'est **pas manipulable telle quelle**
- Il faut **transformer l'objet** en une **séquence binaire**
  - Forme **sérialisée**



Sérialisation d'objets Java (fin)

- Les instances d'une classe ne sont sérialisables que si :
  - La classe implémente `Serializable` ou `Externalizable`
    - Ces interfaces sont vides (définies pour rendre explicite la sérialisation)
  - **Tous les attributs sont serialisables**
- `ObjectOutputStream` : flux (encapsulant) **de sérialisation**
  - Fournit la méthode `writeObject(Object)` qui **construit la forme sérialisée et l'écrit dans le flux de sortie**
- `ObjectInputStream` : flux (encapsulant) **de désérialisation**
  - Fournit la méthode `Object readObject()` qui **lit la forme sérialisée sur le flux d'entrée et retourne la référence de l'objet reconstruit**

PrintStream

- `PrintStream`, étend `FilterOutputStream`
- Ecriture de **représentations textuelles de valeurs de types primitifs** dans un flux
  - Ecrire l'entier 32 correspond à écrire les caractères '3' et '2'

Constructeurs de la classe java.io.PrintStream		
public	PrintStream(OutputStream os)	*Création d'un flux permettant d'écrire des représentations textuelles de valeurs de types primitifs via le flux d'écriture d'octets os.*
public	PrintStream(OutputStream os, boolean auto)	*Idem précédente, avec possibilité de court-circuiter le tampon d'écriture s'il existe.*
public	PrintStream(OutputStream os, boolean auto, String e)	*Idem précédente, en spécifiant le nom du jeu de caractères à utiliser. Soulève `UnsupportedEncodingException` si le jeu de caractères spécifié n'est pas disponible.*

PrintStream (suite)

Méthodes de la classe java.io.PrintStream		
public void	print(boolean b)	*Ecriture de la représentation textuelle d'une valeur booléenne.*
public void	print(int i)	*Ecriture de la représentation textuelle d'une valeur entière.*
public void	print(long l)	*Ecriture de la représentation textuelle d'une valeur de type entier long.*
public void	print(float f)	*Ecriture de la représentation textuelle d'une valeur flottante.*
public void	print(Object o)	*Ecriture de la représentation textuelle d'un objet (écrite la chaîne de caractère retournée par `o.toString()`.*
public void	print(String s)	*Ecriture d'une chaîne de caractères.*

- variantes `println(...)` : écriture puis saut de ligne

PrintStream (fin)

- Exemple : journalisation dans un fichier texte

```
public static void main(String[] args)
// args[0] : chemin complet du fichier
{
   PrintStream ps = null;
   try
   {
      ps = new PrintStream(new FileOutputStream(args[0]));
   }
   catch (FileNotFoundException e) {...}
   try
   {
      long temps = System.currentTimeMillis();
      ps.println("démarrage de l'application ");
      Thread.sleep((int)(Math.random()*10000));
      temps = System.currentTimeMillis() - temps;
      ps.print("l'application a fait une pause de ");
      ps.print(temps);
      ps.println(" ms");
      ps.close();
   }
   catch (Exception e) {...}
}
```

Entrée/sortie/erreur standards

- Classe `java.lang.System`

Attributs de la classe `java.lang.System`		
public static PrintStream	err	*Référence du flux d'erreur standard.*
public static InputStream	in	*Référence du flux d'entrée standard.*
public static PrintStream	out	*Référence du flux de sortie standard.*

Méthodes de la classe `java.lang.System`		
public static void	setErr(PrintStream ps)	*Redirection du flux d'erreur standard vers* ps.
public static void	setIn(InputStream is)	*Redirection du flux d'entrée standard vers* is.
public static void	setOut(PrintStream ps)	*Redirection du flux de sortie standard vers* ps.

Reader

- **Flux de lecture de caractères**
  - Caractère = char = 16 bits = code **Unicode UTF-16**
- Méthodes communes avec `InputStream`
  - `markSupported`, `mark`, `reset`, `skip`, `close`

Méthodes de la classe **abstraite** `java.io.Reader`		
public boolean	ready()	*Test de disponibilité de caractères. Soulève `IOException` en cas de problème.*
public int	read()	*Lecture **bloquante** d'un caractère. Retourne un entier contenant le code du caractère lu, ou -1 si la fin du flux a été atteinte. Soulève `IOException` en cas de problème.*
public int	read(char[] c)	*Lecture **bloquante** d'une suite de caractères (d'au plus `c.length`). Stocke les caractères lus dans le tableau c et retourne le nombre ou -1 si la fin du flux a été atteinte. Soulève `IOException` en cas de problème.*
public abstract int	read(char[] c, int off, int l)	*Idem précédente mais stockage dans un sous-tableau.*

Writer

- **Flux d'écriture de caractères**
- Méthodes communes avec `OutputStream`
  - `flush`, `close`

Méthodes de la classe **abstraite** `java.io.Writer`		
public void	write(int c)	*Ecriture d'un caractère dans le flux. Soulève `IOException` en cas de problème.*
public void	write(char[] c)	*Ecriture d'une suite de caractères dans le flux. Soulève `IOException` en cas de problème.*
public abstract void	write(char[] c, int off, int l)	*Idem précédente, mais écriture d'un sous-tableau.*
public void	write(String s)	*Ecriture d'une chaîne de caractères dans le flux. Soulève `IOException` en cas de problème.*
public void	write(String s, int off, int l)	*Idem précédente, mais écriture d'une sous-chaîne.*

Flux primitifs

- Lire et écrire des caractères **en mémoire** (i.e. dans/vers un `char[]`)
  - `CharArrayReader`, `CharArrayWriter`
- Lire et écrire des caractères dans/vers une **chaîne de caractères**
  - `StringReader`, `StringWriter`
- Lire et écrire des caractères dans/vers un **tube de communication**
  - `PipedReader`, `PipedWriter`

Flux encapsulants

- Lire et écrire des caractères **à travers un tampon**
  - `BufferedReader`, `BufferedWriter`
    - Possibilité de **lire/écrire des lignes de texte**
- Lire des caractères avec **possibilité de retour arrière**
  - `PushbackReader`
- Lire et écrire des caractères à travers un **flux binaire**
  - `InputStreamReader`, `OutputStreamWriter`
    - Gestion de l'**encodage des caractères** (jeu de caractères)
- Lire et écrire des caractères dans/vers un **fichier**
  - `FileReader`, `FileWriter`
    - Raccourci pour `XStreamReader(FileXStream(...))`, ...

Exemple

- Exemple : Lecture de valeurs sur l'entrée standard

```
public static void printUserParams()
{
   InputStreamReader isr = new InputStreamReader(System.in, "US-ASCII");

   BufferedReader br = new BufferedReader(isr);

   String s = null;
   try
   {
      s = br.readLine();
      int i = Integer.parseInt(s);
      System.out.println(i);
      isr.close();
      br.close();
   }
   catch (IOException e) {...}
}
```