

# Programmation Orientée Objet(Java)

GI BAC II  
2020-2021

# Programme

- Une suite d'instructions
- Exemple:
  1. Faire A (un calcul)
  2. Faire B
  3. Tester une condition: si satisfaite aller à 2, sinon, aller à 4
  4. Faire C
- Un programme est écrit dans un langage
  - Langage machine (add 12, ...): bas niveau
  - Langage haut niveau
    - Procédural
      - C, Basic, Cobol, Pascal, Fortran, ...
    - Orienté Objet (OO):
      - Java, VB, C++, ...
    - IA (intelligence artificielle):
      - Prolog, Lisp

# Les étapes

- Écrire un programme dans un langage (e.g. Java)
- Compiler le programme
  - Traduire le programme dans un langage de bas niveau (machine)
  - [éventuellement optimisation]
  - Produire un programme (code) exécutable
- Exécution
  - Charger le programme en mémoire (typiquement en tapant le nom du programme exécutable)
  - Exécution

# Termes

- Programme source, code source
  - Programme écrit dans un langage
- Code machine, code exécutable
  - Programme dans un langage de machine, directement exécutable par la machine
- Compilation (compilateur)
  - Traduire un code source en code exécutable
- Interpréteur
  - Certains langages n'ont pas besoin d'être traduit en code machine
  - La machine effectue la traduction sur la volée (*on the fly*), instruction par instruction, et l'exécute
  - E.g. Prolog, JavaScript

# Programmation

- Syntaxe d'un langage
  - Comment formuler une instruction correcte (grammaire)
- Sémantique
  - Ce que l'instruction réalise
- Erreur
  - de compilation: typiquement reliée à la syntaxe
  - d'exécution: sémantique (souvent plus difficile à détecter et corriger)

# Java

- Langage orienté objet
  - Notions de classes, héritage, ...
- Beaucoup d'outils disponibles (packages)
  - JDK (*Java Development Kit*)
- Historique
  - Sun Microsystems
  - 1991: conception d'un langage indépendant du hardware
  - 1994: browser de HotJava, applets
  - 1996: Microsoft et Netscape commencent à soutenir
  - 1998: l'édition Java 2: plus stable, énorme librairie

# Java

- Compiler un programme en *Byte Code*
  - *Byte code*: indépendant de la machine
  - Interprété par la machine
- *javac programme.java*
  - Génère programme.class
- *java programme*
  - Lance le programme

# Écrire un programme

```
public class Hello
```

Nom de la classe

```
{
```

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

Une méthode

```
{
```

```
    // afficher une salutation
```

commentaire

```
    System.out.println("Hello, World!");
```

Une instruction

```
}
```

```
}
```

- Stocker ce programme dans le fichier **Hello.java**



# Lancer un programme

- Compilation
  - *javac Hello.java*
  - Ceci génère *Hello.class*
- Lancer l'exécution
  - *java Hello*
- Résultat de l'exécution

Hello, World!

# Éléments de base dans un programme

- mots réservés: public class static void
- identificateurs: args Hello main String System out println
  - main String System out println: ont une fonction prédéfinie
- littéral: "Hello World!"
- ponctuation: { accolade } [ crochet ] ( parenthèse )
- Commentaires
  - // note importante pour comprendre cette partie du code
  - /\* ... commentaires sur plusieurs lignes
  - \*/

# Classe

- Un programme en Java est défini comme une classe
- Dans une classe:
  - attributs, méthodes
- L'en-tête de la classe
  - `public class NomDeClasse`
    - *public* = tout le monde peut utiliser cette classe
    - *class* = unité de base des programmes OO
- Une classe par fichier
- La classe `NomDeClasse` doit être dans le fichier `NomDeClasse.java`
- Si plus d'une classe dans un fichier *.java*, *javac* génère des fichiers *.class* séparés pour chaque classe

# Classe

- Le corps

{

...

}

- Contient les attributs et les méthodes
  - Attributs: pour stocker les informations de la classe
  - Méthodes: pour définir ses comportement, ses traitements, ...
- Conventions et habitudes
  - nom de classe: **NomDeClasse**
  - indentation de { }
  - indentation de ...
  - Les indentations correctes ne seront pas toujours suivies dans ces notes pour des raisons de contraintes d'espace par PowerPoint...

# Méthode: en-tête

- L'en-tête:

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

- *main*: nom de méthode
- *void*: aucune sortie (ne retourne rien)
- *String[] args*: le paramètre (entrée)
  - *String[]*: le type du paramètre
  - *args*: le nom du paramètre

- Conventions

- nomDeParametre
- nomDeMethode
- nomDAttributs
- nomDObjet

# Méthode: corps

- Le corps:

```
{  
    // afficher une salutation  
    System.out.println("Hello, World!");  
}
```

- contient une séquence d'instructions, délimitée par { }
  - // afficher une salutation : commentaire
  - System.out.println("Hello, World!"): appel de méthode
- les instructions sont terminées par le caractère ;

# Méthode: corps

- En général:

`nomDObjet.nomDeMethode(<liste des paramètres>)`

- `System.out`: l'objet qui représente le terminal (l'écran)
- `println`: la méthode qui imprime son paramètre (+ une fin de ligne) sur un *stream* (écran)

`System.out.println("Hello, World!");`

- `"Hello, World!"`: le paramètre de `println`

- La méthode `main`

- `"java Hello"` exécute la méthode *main* dans la classe *Hello*
- *main* est la méthode exécutée automatiquement à l'invocation du programme (avec le nom de la classe) qui la contient

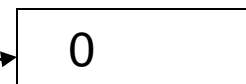
# Variable

- Variable: contient une valeur
  - Nom de variable
  - Valeur contenue dans la variable
  - Type de valeur contenue
    - *int*: entier, *long*: entier avec plus de capacité
    - *Integer*: classe entier, avec des méthodes
    - *float*: nombre réel avec point flottant, *double*: double précision
    - *String*: chaîne de caractères ("Hello, World!")
    - *char*: un caractère en Unicode ('a', '\$', 'é', ...)
    - *boolean*: true/false
- Définition générale  
Type nomDeVariable;

Exemple: `int age;`

Type: int

Nom: age





# Modifier la valeur

- Affecter une valeur à une variable
- E.g. `age = 25;`

Type: int

Nom: age



25

- Erreur si: `age = "vingt cinq";`
  - Type de valeur incompatible avec la variable

# Condition et test

- Une condition correspond à vrai ou faux
- E.g. (*age* < 50)
- Tester une condition:
  - if condition A; else B;*
  - si *condition* est satisfaite, alors on fait *A*;
  - sinon, on fait *B*
  - E.g. *if (age < 65)*
    - System.out.println("jeune");*
    - else*
    - System.out.println("vieux");*

# Tests

- Pour les valeurs primitives (int, double, ...)
  - `x == y` : x et y ont la même valeur?
  - `x > y`, `x >= y`, `x != y`, ...
  - Attention: (`==` `!=` `=`)
- Pour les références à un objet
  - `x == y` : x et y pointent vers le même objet?
  - `x.compareTo(y)`: retourne -1, 0 ou 1 selon l'ordre entre le contenu des objets référés par x et y

# Un exemple de test

```
public class Salutation
{
    public static void main(String[] args)
    {
        int age;
        age = Integer.parseInt(args[0]);
        // afficher une salutation selon l'age
        System.out.print("Salut, le ");
        if (age < 65)
            System.out.println("jeune!");
        else
            System.out.println("vieux!");
    }
}
```

args[0]: premier argument  
après le nom

Integer.parseInt(args[0]):  
reconnaître et transmettre  
la valeur entière qu'il  
représente

print: sans retour à la ligne  
println: avec retour à la ligne

- Utilisation:

- java Salutation 20  
    Salut le jeune!
- java Salutation 70  
    Salut le vieux!

// ici, args[0] = "20"

### Attention:

un ; après le for( ), itère sur la condition, et 'somme' ne sera incrémentée qu'une seule fois

# Boucle

### Attention:

'i' n'est déclarée ici qu'à l'intérieur de la boucle for

- Pour traiter beaucoup de données en série
- Schémas

- Boucle *for*

```
int somme = 0;
```

```
for (int i = 0; i<10; i++) somme = somme + i;
```

- Boucle *while*

```
int somme = 0;
```

```
int i = 0;
```

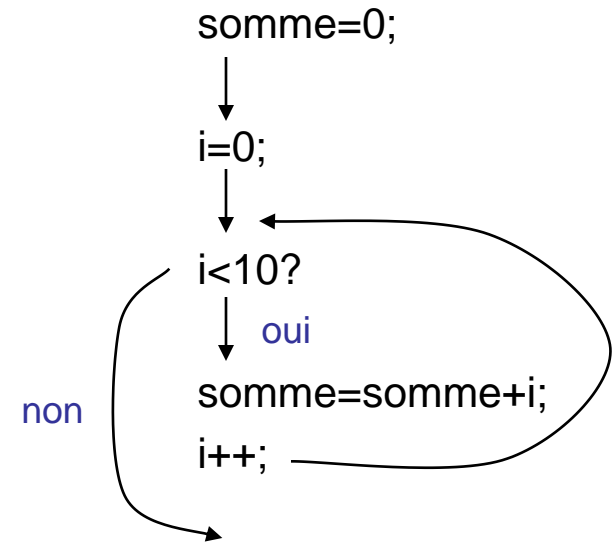
```
while (i<10) { somme = somme + i;
```

```
    i++;
```

```
}
```

- Que font ces deux boucles?

## Schéma d'exécution



i:	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10	
	↓	↓								↓		
somme:	0	→ 0,	→ 1,	3,	6,	10,	15,	21,	28,	36,	→ 45,	sortie

# Boucle

- `do A while (condition)`

- Faire *A* au moins une fois
- Tester la condition pour savoir s'il faut refaire *A*

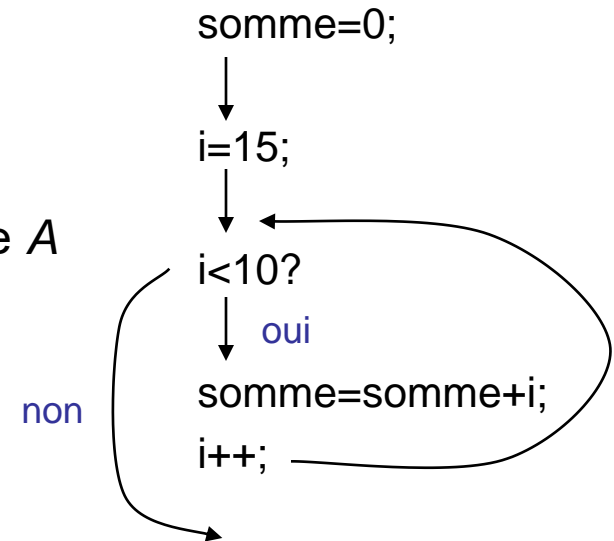
```
int somme = 0;  
int i = 15;  
while (i<10) { somme = somme + i;  
              i++;  
            }
```

somme = 0

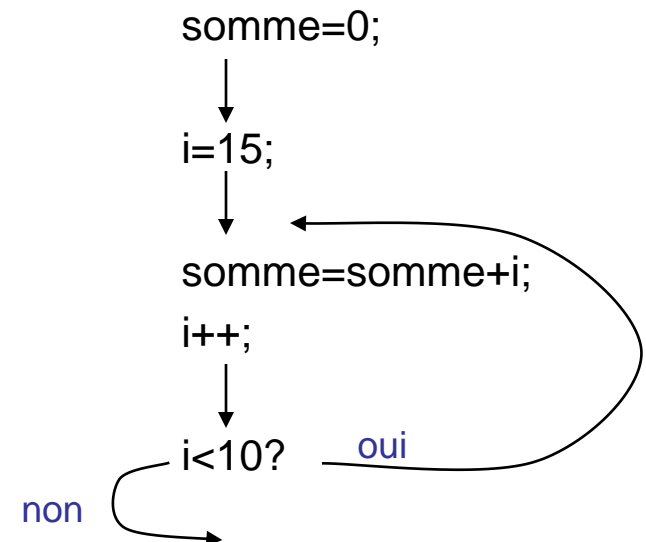
```
int somme = 0;  
int i = 15;  
do { somme = somme + i;  
    i++;  
  }  
while (i<10)
```

somme = 15

## Schéma d'exécution



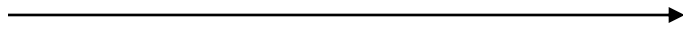
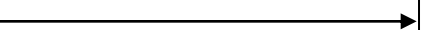
## Schéma d'exécution



# Exemple

- Calcul des intérêts
- Étant donné le solde initial, le solde souhaité et le taux d'intérêt, combien d'années seront nécessaires pour atteindre le solde souhaité
  - au lieu d'utiliser une formule, on simule le calcul
- Algorithme (pseudocode):
  1. ans = 0;
  2. WHILE solde n'atteint pas le solde souhaité
  3. incrémenter ans
  4. ajouter l'intérêt au solde

# Programme

```
public void nombreAnnees (double balance, double targetBalance,  
    double rate ) {  
    int years = 0;  
    while (balance < targetBalance) {  
        years++;  years = years + 1;  
        double interest = balance * rate;  
        balance += interest;  balance = balance + interest;  
    }  
    System.out.println(years + " years are needed");  
}
```

Appel de la méthode:

`nombreAnnees(1000, 1500, 0.05)`

Résultat:

`56 years are needed`



# Factorielle

```
public class Factorielle
```

```
{  
    public static double factorielle(int x) {  
        if (x < 0) return 0.0;  
        double fact = 1.0;  
        while (x > 1) {  
            fact = fact * x;  
            x = x - 1;  
        }  
        return fact;  
    }  
}
```

```
public static void main(String[] args) {  
    int entree = Integer.parseInt(args[0]);  
    double resultat = factorielle(entree);  
    System.out.println(resultat);  
}  
}
```

Si une méthode (ou un attribut, une variable) de la classe est utilisée par la méthode main (*static*), il faut qu'il soit aussi *static*.

# Tableau

## Attention:

Array

a.length

Array list

a.size()

String

a.length()

- Pour stocker une série de données de même nature
- Déclaration

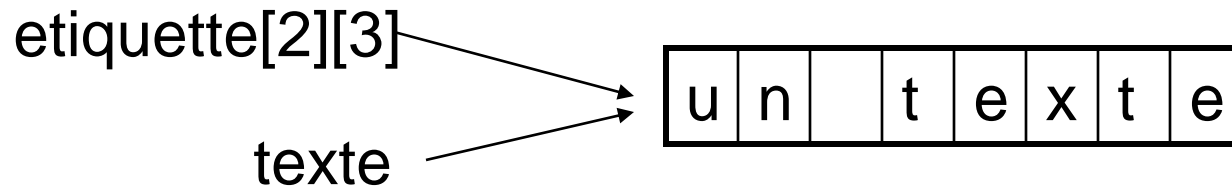
```
int [] nombre; // une série de valeurs int dans le tableau nommé nombre
String [][] etiquette; // un tableau à deux dimensions de valeurs String
```
- Création

```
nombre = new int[10]; // crée les cases nombre[0] à nombre[9]
etiquette = new String[3][5]; // crée etiquette[0][0] à etiquette[2][4]
int[] primes = {1, 2, 3, 5, 7, 7+4}; // déclare, crée de la bonne taille et initialise
```
- Utilisation

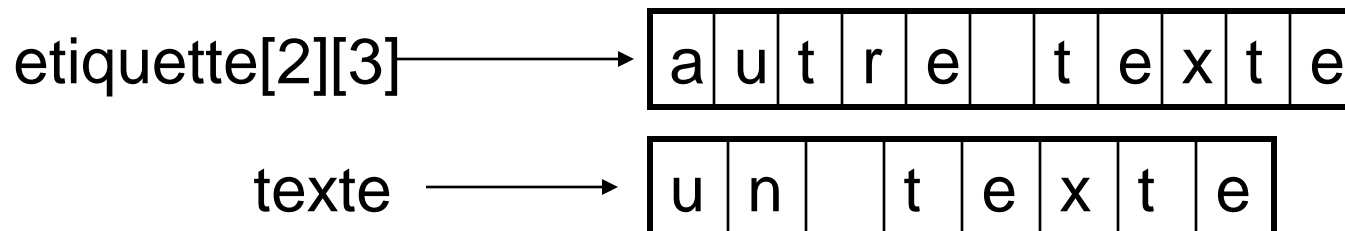
```
nombre[0] = 4;
for (int i=1; i<nombre.length; i++) nombre[i]=nombre[i]+1;
etiquette[2][3] = "un texte";
String texte = etiquette[2][3];
```

# String

- Structure à deux parties:
  - En-tête: nom, longueur, ...
  - Corps: les caractères



- `String texte = etiquette[2][3];`
- Le contenu du corps ne peut pas être changé, une fois qu'il est créé (*String* est immuable)
- Par contre, on peut pointer/référencer `etiquette[2][3]` à un autre corps: `etiquette[2][3] = "autre texte";`



# Classe et Objet

- Classe: moule pour fabriquer des objets
- Objet: élément concret produit par le moule

- Définition de classe:

<pre>class NomClasse {     Attributs;     Méthodes; }</pre>	<pre>class Personne {     String nom;     int AnneeNaissance;     public int age() {...} }</pre>
-------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------

- Une classe regroupe un ensemble d'objets (instances)

# Objet

- Structure à deux parties:
  - Référence
  - Corps
- Les étapes
  - Déclaration de la classe (e.g. *Personne*)
  - À l'endroit où on utilise:
    - Déclarer une référence du type de la classe
    - Créer une instance d'objet (*new*)
    - Manipuler l'objet

# Exemple

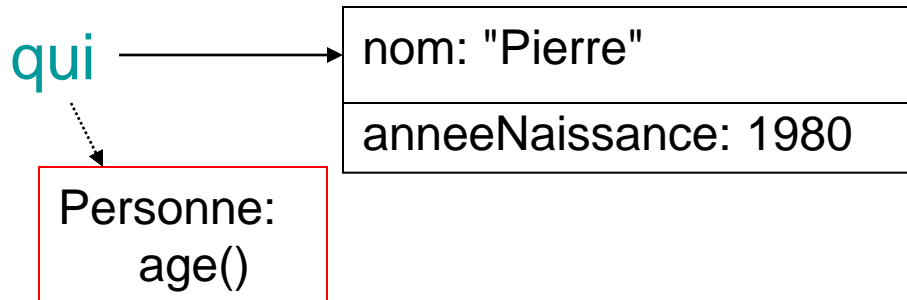
```
public class Personne {  
    public String nom;  
    public int anneeNaissance;  
    public int age() {return 2008 - anneeNaissance; }  
}
```

```
class Utilisation {  
    public static void main(String[] args) {  
        Personne qui;  
        qui = new Personne();  
        qui.nom = "Pierre";  
        qui.anneeNaissance = 1980;  
        System.out.println(qui.age());  
    }  
}
```

Déclaration de référence

Création d'une instance

Manipulation de l'instance  
référée par la référence



# Un autre exemple

```
class Circle {  
    public double x, y; // coordonnées du centre  
    private double r; // rayon du cercle  
    public Circle(double r) {  
        this.r = r;  
    }  
    public double area() {  
        return 3.14159 * r * r;  
    }  
}
```

'r' est inaccessible de l'extérieur de la classe

constructeur

Math.PI

```
public class MonPremierProgramme {  
    public static void main(String[] args) {  
        Circle c; // c est une référence sur un objet de type Circle, pas encore un objet  
        c = new Circle(5.0); // c référence maintenant un objet alloué en mémoire  
        c.x = c.y = 10; // ces valeurs sont stockées dans le corps de l'objet  
        System.out.println("Aire de c :" + c.area());  
    }  
}
```

# Constructeurs d'une classe

- Un constructeur est une façon de fabriquer une instance
- Une classe peut posséder plusieurs constructeurs
- Si aucun constructeur n'est déclaré par le programmeur, alors on a la version par défaut: `NomClasse()`
- Plusieurs versions d'une méthode: surcharge (*overloading*)

```
class Circle {  
    public double x, y; // coordonnées du centre  
    private double r; // rayon du cercle  
    public Circle(double r) {  
        this.r = r;  
    }  
    public Circle(double a, double b, double c) {  
        x = a; y = b; r = c;  
    }  
}
```

*this*: réfère à l'objet courant

```
public class Personne {  
    public String nom;  
    public int anneeNaissance;  
    public int age() { return 2008 - anneeNaissance; }  
}
```

Le constructeur  
`Personne()` est  
déclaré par défaut



# Manipulation des références

```
class Circle {  
    public double x, y; // coordonnées du centre  
    private double r; // rayon du cercle  
    public Circle(double r) {  
        this.r = r;  
    }  
    public Circle(double a, double b, double c) {  
        x = a; y = b; r = c;  
    }  
}
```

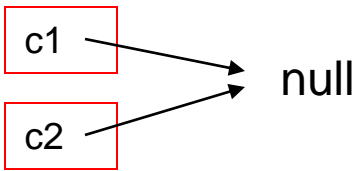
// Dans une méthode, par exemple, main:

```
Circle c1, c2;  
c1 = new Circle(2.0, 3.0, 4.0);  
c2 = c1; // c2 et c1 pointent vers le même objet  
c2.r = c2.r - 1; // l'objet a le rayon réduit  
c1 = new Circle(2.0); // c1 point vers un autre objet, mais c2 ne change pas  
c1.x = 2.0; // on modifie le deuxième objet  
c2 = c1; // maintenant, c2 pointe vers le 2ième objet aussi
```

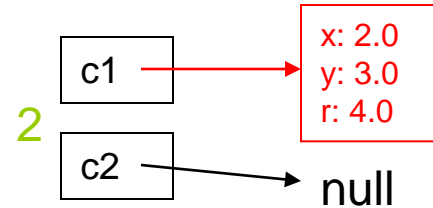
- Que faire du premier objet?
  - Aucune référence ne pointe vers lui
  - L'objet est perdu et inutilisable
  - Ramasse miette (*garbage collector*) va récupérer l'espace occupé par l'objet
- Comparaison des références
  - (c1 == c2): est-ce que c1 et c2 pointent vers le même objet?

# Illustration

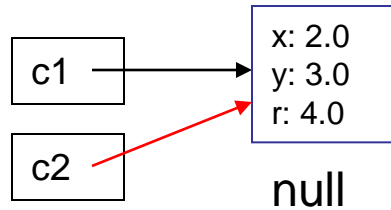
1. Cercle c1, c2; 1



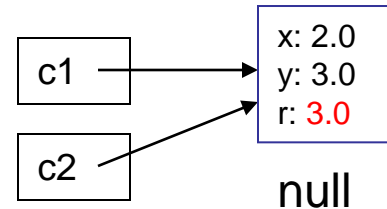
2. c1 = new Cercle(2.0, 3.0, 4.0); 2



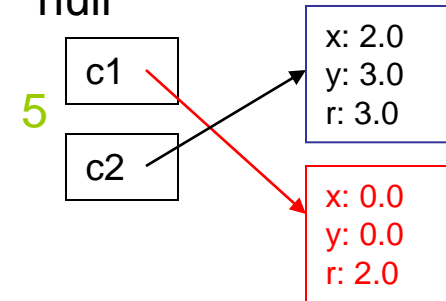
3. c2 = c1; 3



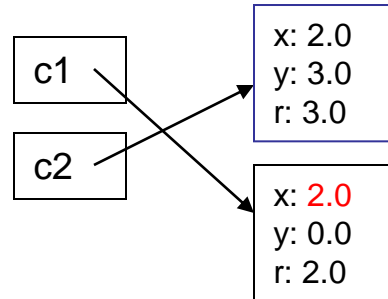
4. c2.r = c2.r - 1; 4



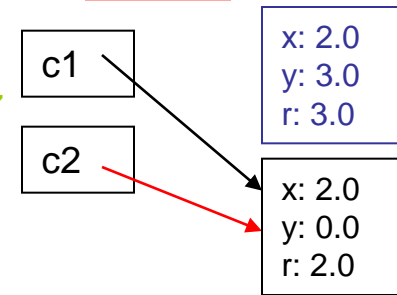
5. c1 = new Cercle(2.0); 5



6. c1.x = 2.0; 6



7. c2 = c1; 7



# Manipulation des objets

- Forme générale

*référence.attribut*: réfère à un attribut de l'objet

*référence.méthode()*: réfère à une méthode de l'objet

- *static*: associé à une classe

- Attribut (variable) statique: si on le change, ça change la valeur pour tous les objets de la classe

- Méthode statique: on la réfère à partir de la classe

- *Classe.méthode*

- E.g. *Math.sqrt(2.6)*: Appel à la méthode *sqrt* de la classe *Math*

- Constante: *Math.PI*

- Dans une classe: *static final float PI = 3.14159265358979;*

- Une constante n'est pas modifiable

# Classes et Héritage

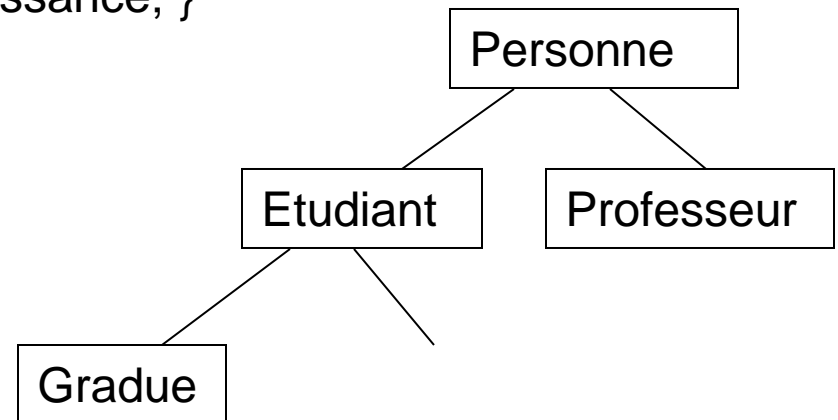
- Héritage
  - Les enfants héritent les propriétés du parent
  - Classe enfant (sous-classe) possède systématiquement les attributs et les méthodes de la classe parent (super-classe)
  - Héritage simple (une seule super-classe au plus)

• E.g.

```
class Personne {  
    String nom;  
    int anneeNaissance;  
    public int age() { return 2008 - anneeNaissance; }  
}
```

```
class Etudiant extends Personne {  
    String [] cours;  
    String niveau;  
    String ecole;  
    ...  
}
```

- Ce qui est disponible dans Etudiant:
  - nom, anneeNaissance, age(),
  - cours, niveau, ecole, ...



# Principe

- Définir les propriétés communes dans la classe supérieure
- Définir les propriétés spécifiques dans la sous-classe
- Regrouper les objets le plus possible
- Les objets d'une sous-classe sont aussi des objets de la super-classe
- La classe dont tous les objets appartiennent:  
*Object*
- Tester l'appartenance d'un objet dans une classe: *instanceof* (e.g. *qui instanceof Etudiant*)

# Exemple

```
public class Ellipse {  
    public double r1, r2;  
    public Ellipse(double r1, double r2) { this.r1 = r1; this.r2 = r2; }  
    public double area() {...}  
}
```

super(r,r): constructeur de la super-classe

```
final class Circle extends Ellipse {  
    public Circle(double r) {super(r, r);}  
    public double getRadius() {return r1;}  
}
```

final assure qu'aucune autre classe n'héritera de Circle

// Dans une méthode

```
Ellipse e = new Ellipse(2.0, 4.0);
```

```
Circle c = new Circle(2.0);
```

```
System.out.println("Aire de e: " + e.area() + ", Aire de c: " + c.area());
```

```
System.out.println((e instanceof Circle)); // false
```

```
System.out.println((e instanceof Ellipse)); // true
```

```
System.out.println((c instanceof Circle)); // true
```

```
System.out.println((c instanceof Ellipse)); // true (car Circle dérive de Ellipse)
```

```
e = c;
```

```
System.out.println((e instanceof Circle)); // true
```

```
System.out.println((e instanceof Ellipse)); // true
```

```
int r = e.getRadius(); // erreur: méthode getRadius n'est pas trouvée dans la classe Ellipse
```

```
c = e; // erreur: type incompatible pour = Doit utiliser un cast explicite
```

# Casting

- La classe de la référence détermine ce qui est disponible (a priori)

- E.g.

```
class A {  
    public void meth() { System.out.println("Salut"); }  
}  
class B extends A {  
    int var;  
}
```

```
A a = new A();
```

```
B b = new B();
```

```
a.meth(); // OK
```

```
b.meth(); // OK, héritée
```

```
b.var = 1; // OK
```

```
a.var = 2; // erreur
```

```
a = b;
```

```
a.var = 2; // erreur: var n'est a priori pas disponible pour une classe A
```

```
((B) a).var = 2; // OK, casting
```

- *Casting*: transforme une référence d'une super-classe à celle d'une sous-classe
- Condition: l'objet référencé est bien de la sous-classe

# Surcharge de méthode

```
class A {  
    public void meth() {System.out.println("Salut"); }  
}  
class B extends A {  
    public void meth(String nom) {  
        System.out.println("Salut" +nom);  
    }  
}
```

- Dans la sous-classe: une version additionnelle
  - Signature de méthode: nom+type de paramètres
  - Surcharge: créer une méthode ayant une autre signature



# Overriding: écrasement

- Par défaut, une méthode est héritée par une sous-classe
- Mais on peut redéfinir la méthode dans la sous-classe (avec la même signature)
- Les objets de la sous-classe ne possèdent que la nouvelle version de la méthode
- E.g.

```
class A {  
    public void meth() {System.out.println("Salut");}  
}  
class B extends A {  
    public void meth() {System.out.println("Hello");}  
}
```

```
A a = new A();
```

```
B b = new B();
```

```
a.meth(); // Salut
```

```
b.meth(); // Hello
```

```
a = b; // a réfère à un objet de classe B
```

```
a.meth(); // Hello. Même si la référence est de classe A, l'objet est de classe B
```

# Classe abstraite

- Certains éléments peuvent être manquants dans une classe, ou la classe peut être trop abstraite pour correspondre à un objet concret
- Classe abstraite
  - Une classe non complétée ou une classe conceptuellement trop abstraite
    - Classe *Shape*
      - on ne connaît pas la forme exacte, donc impossible de créer un objet
      - cependant, on peut savoir que chaque *Shape* peut être dessinée

```
abstract class Shape {  
    abstract void draw();  
}
```

# Interface

- Interface

- Un ensemble de méthodes (comportements) exigées
- Une classe peut se déclarer conforme à (implanter) une interface: dans ce cas, elle doit implanter toutes les méthodes exigées

- E.g.

```
public abstract interface Inter {  
    public abstract int carre(int a);  
    public abstract void imprimer();  
}  
class X implements Inter {  
    public int carre(int a) { return a*a; }  
    public void imprimer() {System.out.println("des informations"); }  
}
```

# Example

```
abstract class Shape { public abstract double perimeter(); }
interface Drawable { public void draw(); }
class Circle extends Shape implements Drawable, Serializable {
    public double perimeter() { return 2 * Math.PI * r ; }
    public void draw() {...}
}
class Rectangle extends Shape implements Drawable, Serializable {
    public double perimeter() { return 2 * (height + width); }
    public void draw() {...}
}
...
Drawable[] drawables = {new Circle(2), new Rectangle(2,3),
                        new Circle(5)};
for(int i=0; i<drawables.length; i++)
    drawables[i].draw();
```

# Utilité de l'interface

- Permet de savoir qu'une classe contient les implantations de certaines méthodes
- On peut utiliser ces méthodes sans connaître les détails de leur implantation
- Souvent utilisée pour des types abstraits de données (e.g. pile, queue, ...)

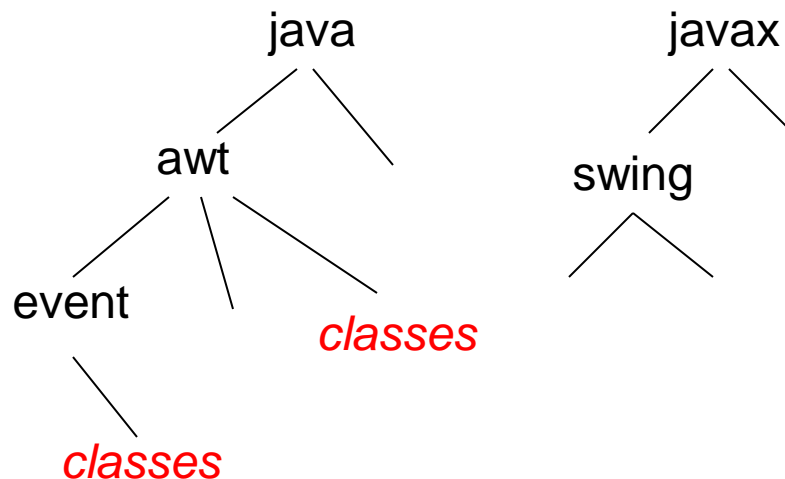
# Package

- On organise les classes et les outils selon leurs fonctionnalités et les objets qu'elles manipulent
- Les classes qui traitent les mêmes objets: *package*
- Exemple:
  - Les classes pour traiter l'interface graphique sont dans le package *awt*

- Organisation des packages

- Hiérarchie

- java.awt
    - java.awt.event
    - javax.swing
    - ...



# Utiliser les packages existants

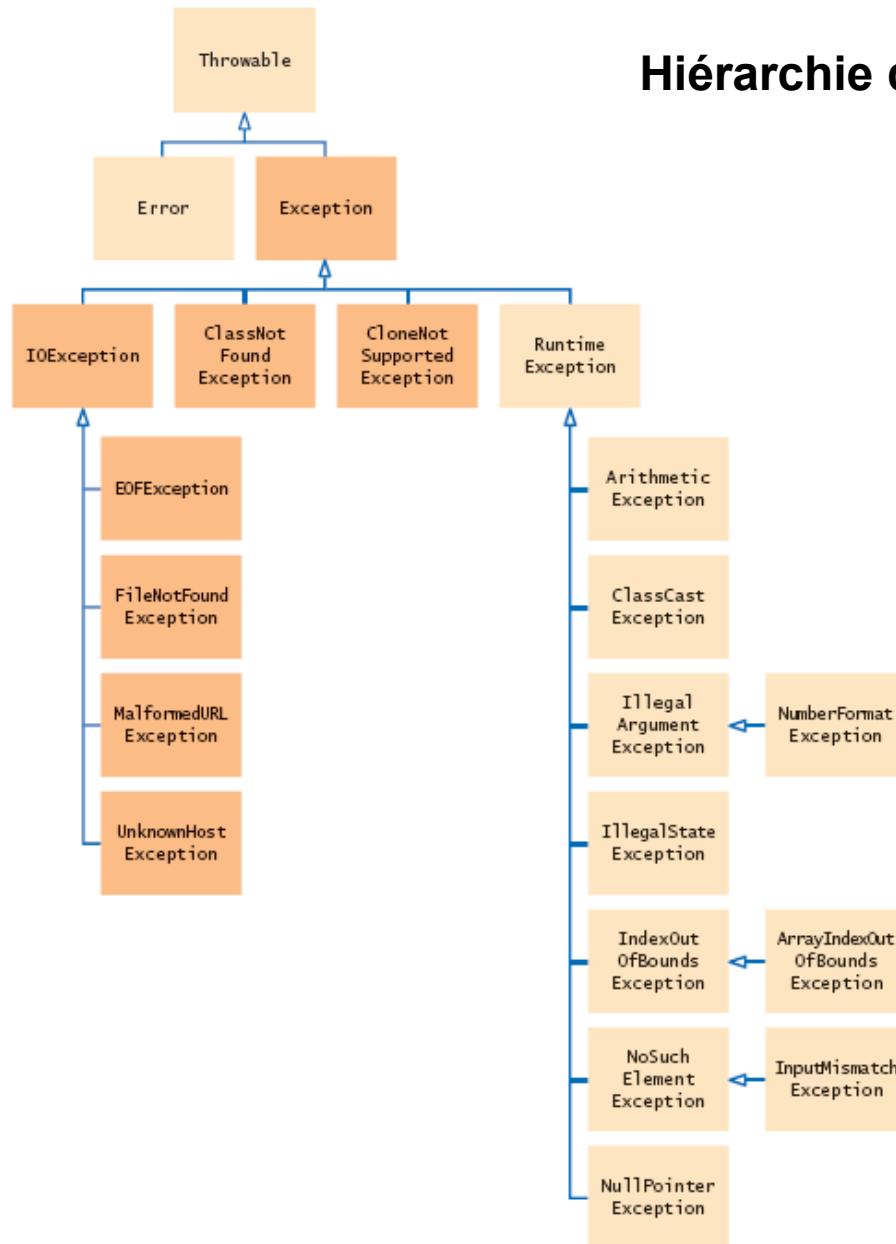
- Au début d'un fichier, importer les classes d'un package
- `import java.awt.*;`
  - Importer toutes les classes du package `java.awt` (reliées aux fenêtres)
- `import java.awt.event.*;`
  - Importer toutes les classes du package `java.awt.event` (reliées au traitement d'événements)

# Exception

- Quand un cas non prévu survient, il est possible de le capter et le traiter par le mécanisme d'*Exception*
  - Si on capte et traite une exception, le programme peut continuer à se dérouler
  - Sinon, le programme sort de l'exécution avec un message d'erreur
  - Exemple d'exception: division par 0, ouvrir un fichier qui n'existe pas, ...
- Mécanisme de traitement d'exception
  - Définir des classes d'exception
    - *Exception*
      - *IOException*
        - » *EOFException*, ...
  - Utiliser *try-catch* pour capter et traiter des exceptions



# Hiérarchie des classes d'exceptions

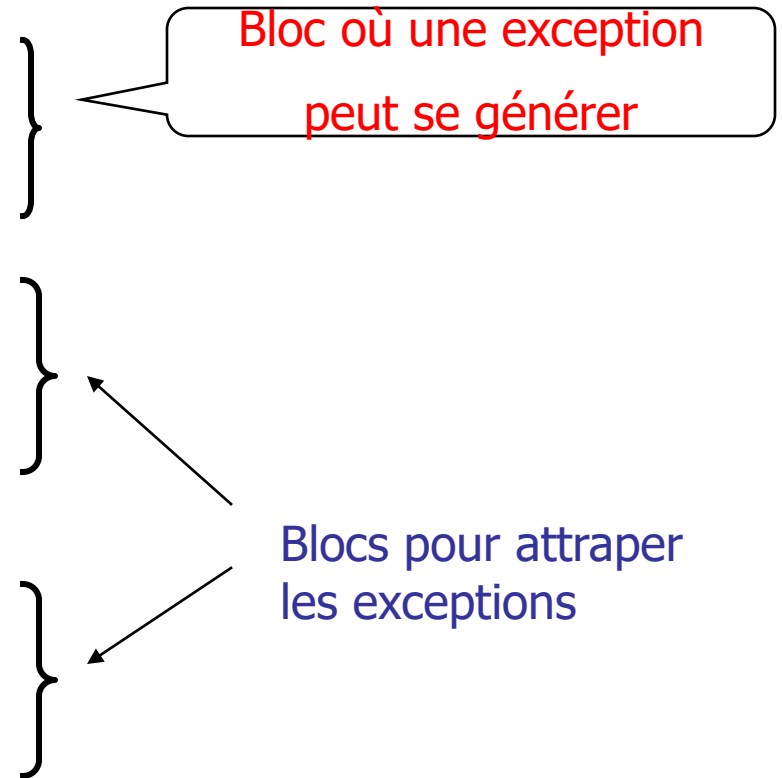


**Figure 1** The Hierarchy of Exception Classes

# Attraper (*catch*) une exception

- Attraper une exception pour la traiter

```
try {  
    statements  
    :  
    :  
} catch (ExceptionClass1 object) {  
    statements  
    :  
    :  
} catch (ExceptionClass2 object) {  
    statements  
    :  
    :  
} ...
```



# Exemple

```
public static void ouvrir_fichier(String nom) {  
    try {  
        input = new BufferedReader(new FileReader(nom));  
    }  
    catch (IOException e) {  
        System.err.println("Impossible d'ouvrir le fichier d'entree.\n" +  
                           e.toString());  
        System.exit(1);  
    }  
}
```

ouverture d'un fichier
---------------------------

*try*: on tente d'effectuer des opérations

*catch*: si une exception de tel type survient au cours, on la traite de cette façon

# finally

- Souvent combiné avec catch

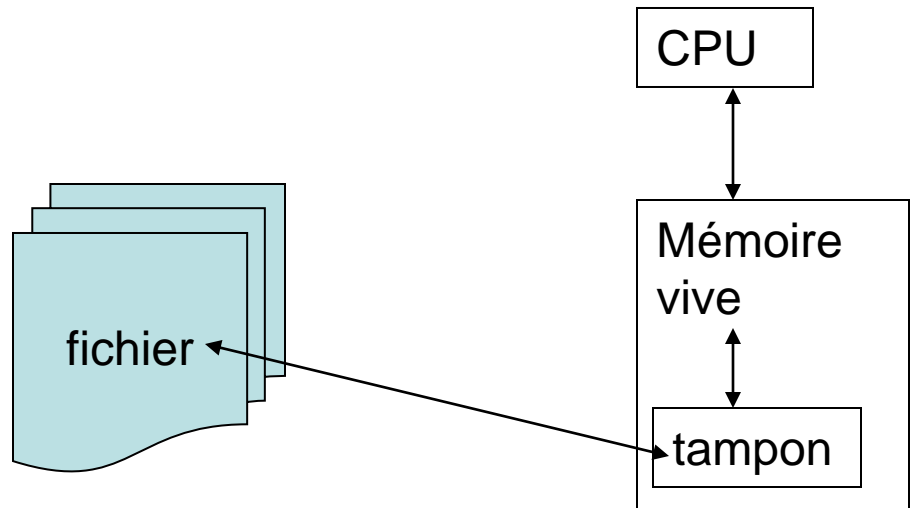
```
try
{
    statements
    ...
}
catch (ExceptionClass exceptionObject)
{
    statements
    ...
}
finally
{
    statements
    ...
}
```

Même si une exception est  
attrapée, *finally* sera toujours  
exécuté

Utile pour s'assurer de certaine  
sécurité (*cleanup*)

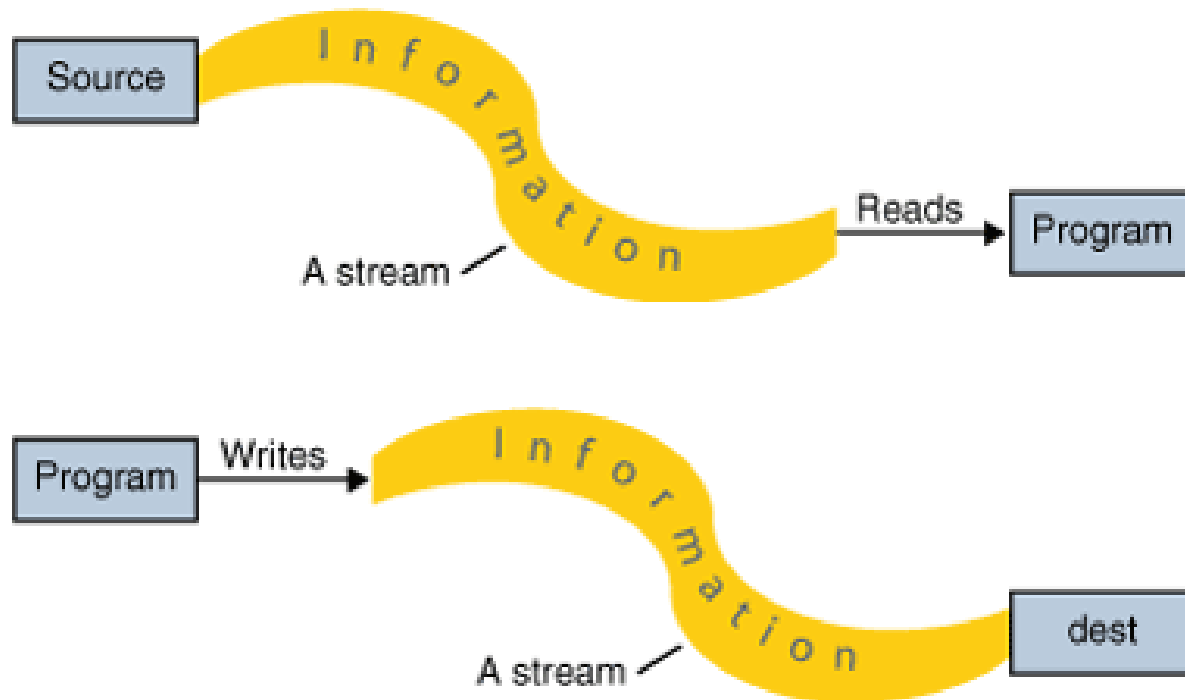
# Fichier

- Unité de stockage des données, sur disque dur
- Stockage permanent (vs. en mémoire vive)
- Un fichier contient un ensemble d'enregistrements
- Traitement



# Fichier en Java

- *Stream*: une suite de données (octets ou caractères)



# Opérations typiques

- Lecture:

- Ouvrir un *stream*
- Lire tant qu'il y a des données
- Fermer le *stream*

Établir un canal de communication

- Écriture

- Ouvrir un *stream* (ou créer un fichier)
- Écrire des données tant qu'il y en a
- Fermer le *stream*

Relâcher les ressources allouées

Écrire ce qu'il est dans le tampon, et relâcher les ressources allouées

# Exemple

```
public static void main(String[] args) {
    ouvrir_fichier("liste_mots");
    traiter_fichier();
    fermer_fichier();
}

public static void ouvrir_fichier(String nom) {
    try {
        input = new BufferedReader(
            new FileReader(nom));
    }
    catch (IOException e) {
        System.err.println("Impossible
d'ouvrir le fichier d'entree.\n" +
e.toString());
        System.exit(1);
    }
}
```

```
public static void traiter_fichier() {
    String ligne;
    try { // catch EOFException

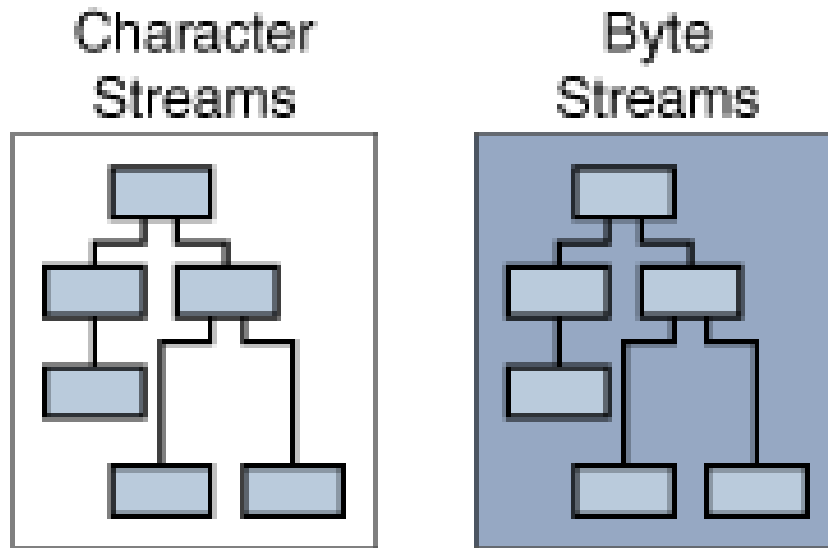
        ligne = input.readLine();
        while (ligne != null) {
            System.out.println(ligne);
            ligne = input.readLine();
        }
    }

    public static void fermer_fichier() {
        try {
            input.close();
            System.exit(0);
        }
        catch (IOException e) {
            System.err.println("Impossible de
fermer les fichiers.\n" + e.toString());
        }
    }
}
```



# Deux unités de base

- Caractère (2 octets=16 bits) ou octet (8 bits)
- Deux hiérarchies de classes similaires (mais en parallèle)

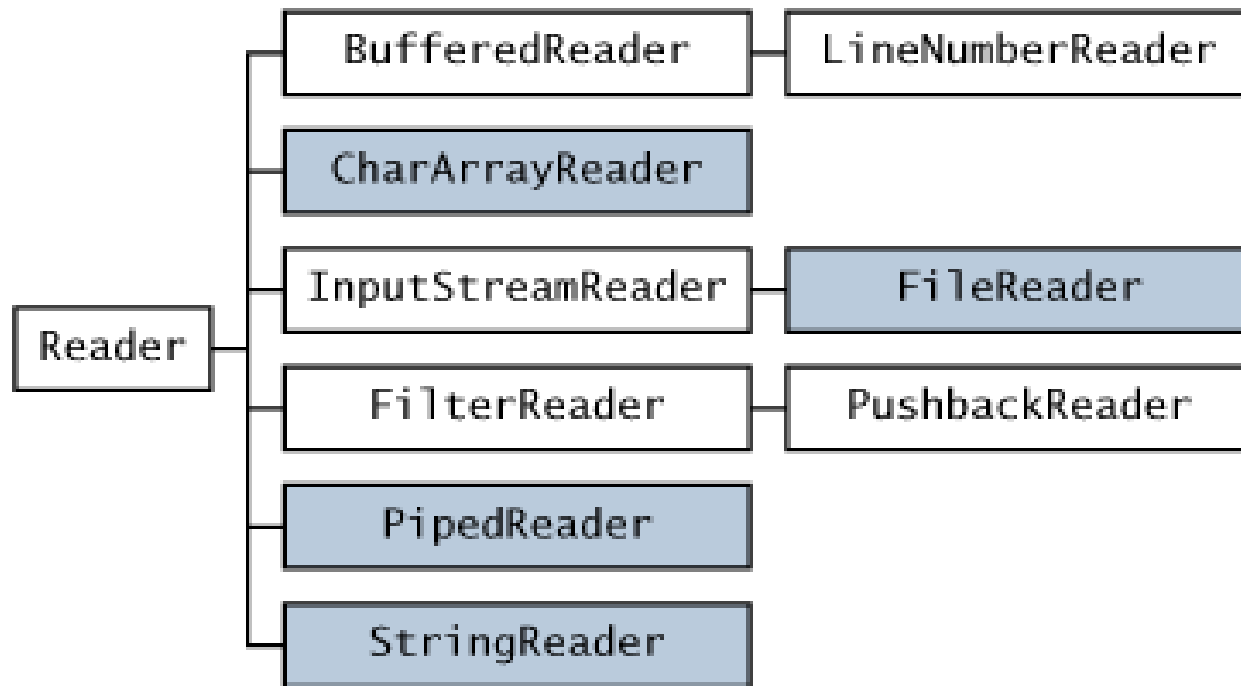


# Hiérarchies

- En haut des hiérarchies pour *stream* de caractères: 2 classes abstraites
- *Reader*
  - java.lang.Object
    - java.io.Reader
- *Writer*
  - java.lang.Object
    - java.io.Writer
- Implémentent une partie des méthodes pour lire et écrire des caractères de 16 bits (2 octets)

# Hiérarchie de *stream* de caractères

- Les sous-classes de *Reader*



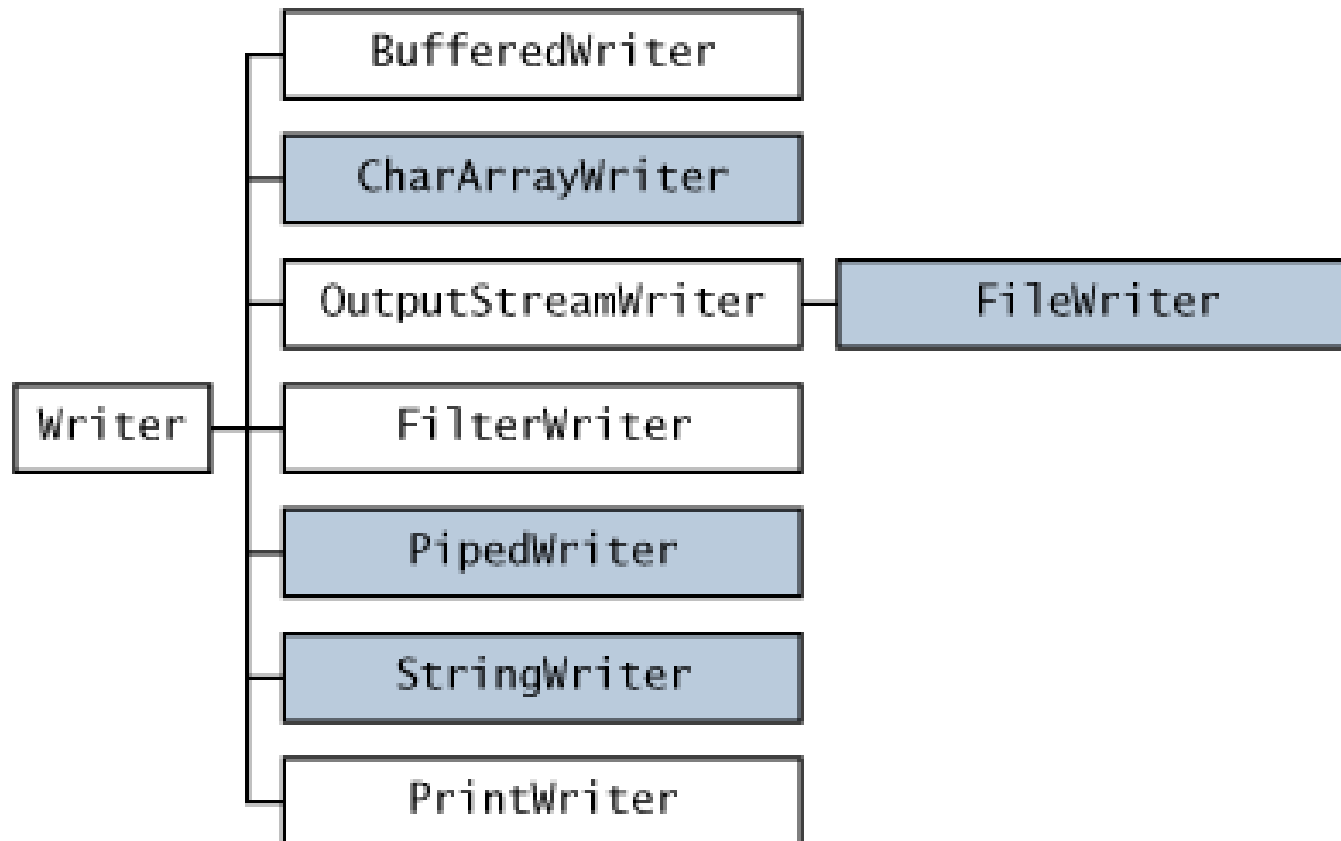
■ simple

□ pré-traitement

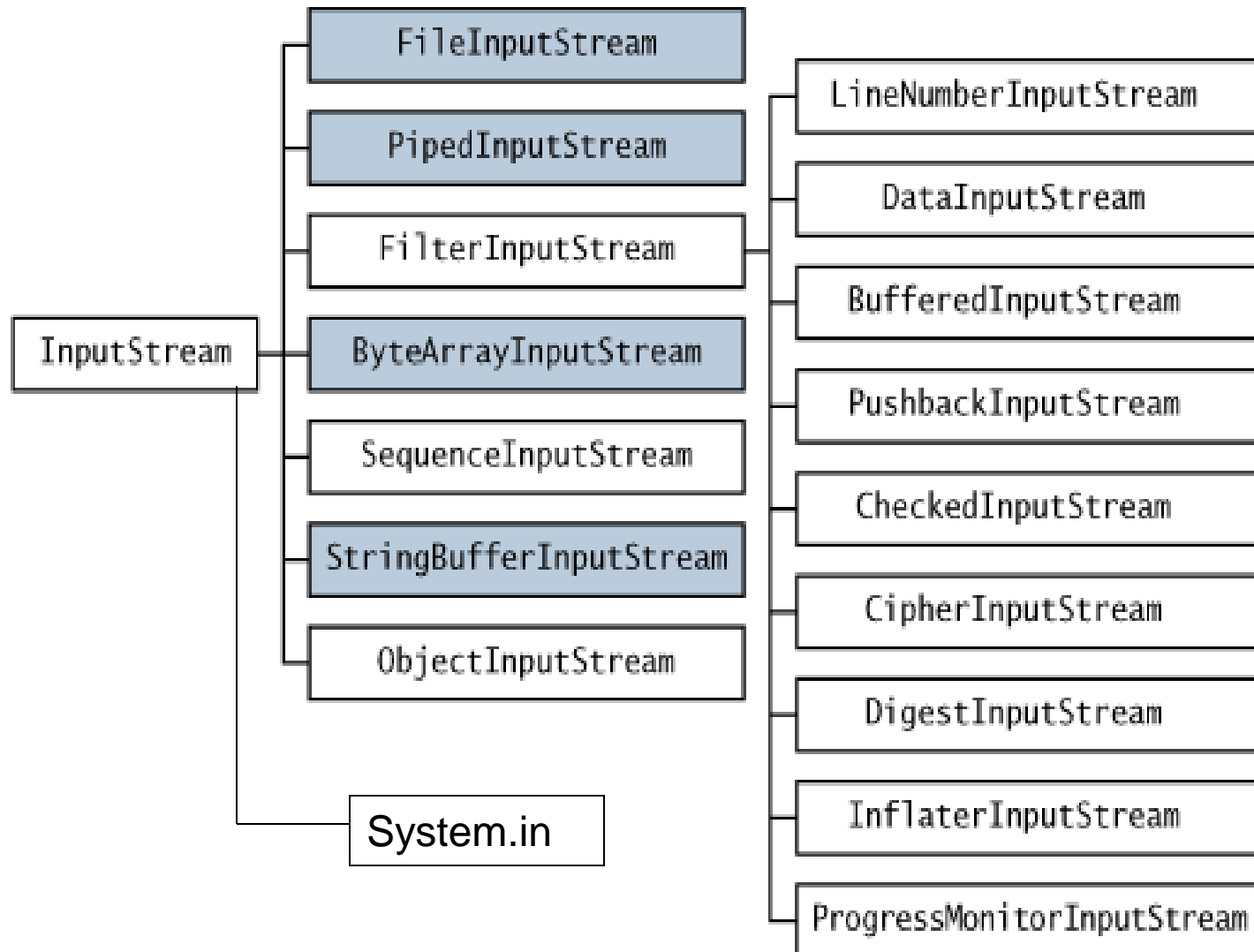
- Chaque sous-classe ajoute des méthodes

# Hiérarchie de *stream* de caractères

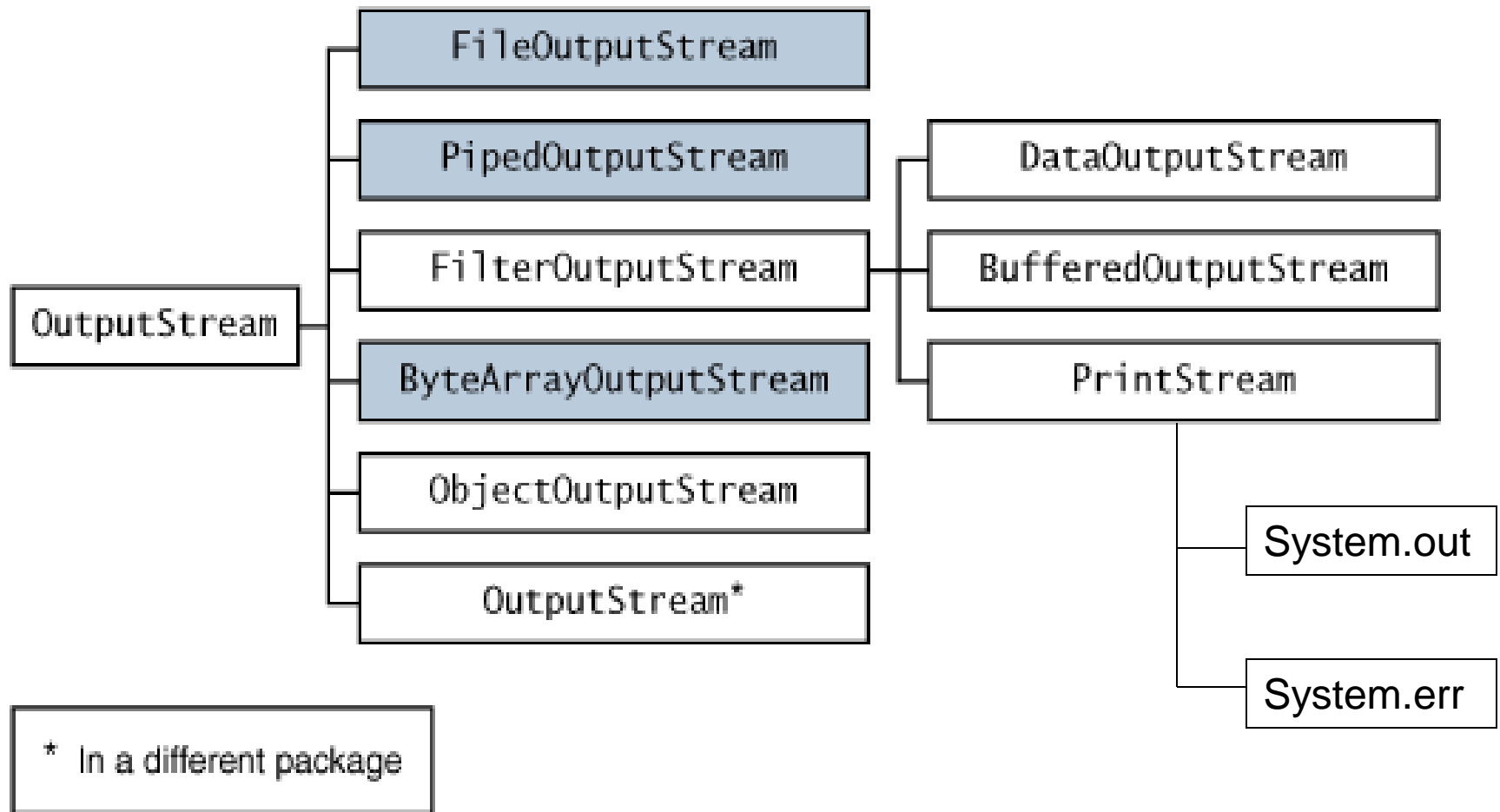
- Les sous-classes de *Writer*



# Hiérarchies *byte stream*



# Hiérarchie de *byte stream*



# Exemple

- Utiliser FileReader et FileWriter
- Méthodes simples disponibles:
  - `int read()`, `int read(CharBuffer [])`, `write(int)`, ..
  - Exemple: copier un fichier caractère par caractère (comme un int)

```
import java.io.*;
public class Copy {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("farrago.txt");
        File outputFile = new File("outagain.txt");

        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;
        while ((c = in.read()) != -1) out.write(c);

        in.close();
        out.close();
    }
}
```

Fin de fichier: -1

- Méthodes limitées

# Augmenter les possibilités: *wrap*

- Créer un *stream* en se basant sur un autre:

```
FileReader in = new FileReader(new File("farrago.txt"));
```

- **Avantage:**

- Obtenir plus de méthodes

- Dans *File*: les méthodes pour gérer les fichiers (`delete()`, `getPath()`, ...) mais pas de méthode pour la lecture
- Dans *FileReader*: les méthodes de base pour la lecture

- Un autre exemple:

```
DataOutputStream out = new DataOutputStream(  
    new FileOutputStream("invoice1.txt"));
```

- *FileOutputStream*: écrire des bytes
- *DataOutputStream*: méthodes pour les types de données de base: `write(int)`, `writeBoolean(boolean)`, `writeChar(int)`, `writeDouble(double)`, `writeFloat(float)`, ...



# Sérialiser

- Convertir un objet (avec une structure) en une suite de données dans un fichier
- Reconvertir du fichier en un objet
- Utilisation: avec **ObjectOutputStream**

```
Employee[] staff = new Employee[3];  
ObjectOutputStream out = new ObjectOutputStream(new  
    FileOutputStream("test2.dat"));  
out.writeObject(staff);  
out.close();
```

# Sérialiser

- Utilité de sérialisation
  - Stocker un objet dans un fichier
  - Créer une copie d'objet en mémoire
  - Transmettre un objet à distance
    - Devient une transmission de *String*