



LES FONDAMENTAUX DE LA PROGRAMMATION JAVA SE

Prof. Bi Tra GOORE

Maître de Conférences, Docteur Ingénieur en Informatique

Institut National Polytechnique Félix Houphouët Boigny

Yamoussoukro – Côte d'Ivoire

bitra.goore@inphb.ci - (+225) 58 50 13 04



Les fondamentaux

Objectif général

A l'issue de ce cours l'apprenant sera capable de:

- Expliquer et appliquer les concepts de base du langage Java
- Utiliser les principales bibliothèques standard (API) de Java
- Ecrire, compiler et exécuter un programme Java en ligne de commande
- Utiliser un IDE pour développer un programme Java



Les fondamentaux

A l'issue de ce cours l'apprenant sera capable de:

- Expliquer et appliquer les concepts de base du langage Java
- Utiliser les principales librairies standard (API) de Java
- Ecrire, compiler et exécuter un programme Java en ligne de commande
- Utiliser un IDE pour développer un programme Java

- 1 Qu'est ce que Java SE ?
- 2 Structure de données et de contrôle de base
- 3 Classes et objets
- 4 Héritage
- 5 Classe abstraite
- 6 Interface
- 7 Notion de paquetage
- 8 Les exceptions
- 9 La gestion des flux d'entrée-Sorties



Les fondamentaux

PLAN

- 1 Qu'est ce que Java SE ?
- 2 Structure de données et de contrôle de base
- 3 Classes et objets
- 4 Héritage
- 5 Classe abstraite
- 6 Interface
- 7 Notion de paquetage
- 8 Exceptions
- 9 La gestion des flux d'entrée-Sorties

Introduction

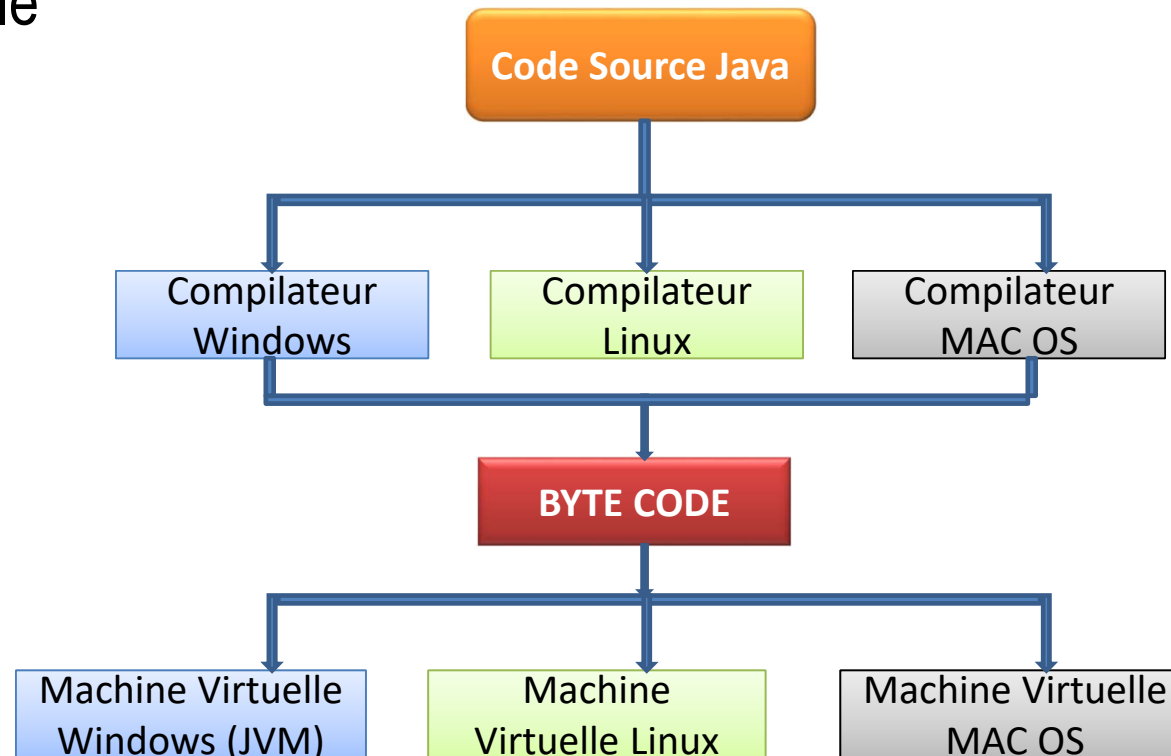
■ Principales caractéristiques

- Plate-forme et langage Java issus d'un projet de *Sun Microsystems* datant de 1990
- Langage orienté objet avec syntaxe similaire à celle du langage C/C++.
- Langage «full object » : tout est objet
- Langage orientée programmation réseau
- Langage multitâche ou multi-activité (multi-thread)
- Langage Portable

Introduction

■ Principales caractéristiques (suite)

- Langage portable



Introduction

■ Environnement Java

- Un langage de programmation
- Une machine virtuelle (JVM)
- Un compilateur
- Un ensemble de classes standards réparties dans des API (Application Programming Interface)
- Un ensemble d'outil (javadoc, jar, etc.)

Introduction

■ Environnement de Développement intégré (IDE)

- Eclipse
- NetBeans
- IntelliJIDE
- JCreator
- BlueJ
- Jbuilder
- jDeveloper
- Visual J++

Introduction

■ Les plateformes JAVA

- JavaCard
 - Applications liées aux cartes à puces et autres SmartCards.
- Java 2 Micro Edition (J2ME) → Java ME (version 8)
 - Applications mobiles
 - Terminaux portables: téléphones, PDAs, TV box, ...
- Java 2 Standard Edition (J2SE) → Java SE (version 8)
 - Postes clients: applications autonomes, applets, ...
 - Bases nécessaires pour JEE
- Java 2 Entreprise Edition (J2EE) → java EE (version 8)
 - Applications serveurs.
 - Solutions pour les entreprises : E-commerce, E-business

Structures de données de base

Structures de contrôle

1. Les types de bases (entier, tableau)
2. Les structures conditionnelles
3. Les structures de boucles

Structure de donnée de base

■ Les types de base

type	signification	taille en octet
char	Caractère Unicode	2
byte	Entier très court	1
short	Entier court	2
int	Entier	4
long	Entier long	8
float	Nombre réel simple	4
double	Nombre réel double	8
boolean	Valeur logique	1

Les tableaux

■ Tableaux unidimensionnels

- Déclaration

```
type [] nomTableau;  
type nomTableau[];
```

- Création à l'aide de l'instruction « new »

```
nomTableau = new type [taille]
```

- Exemple

```
int [] tab;  
Tab = new int [10];
```

Les tableaux

■ Tableaux à plusieurs dimension

- Déclaration

```
Type  [][] nomTableau;  
Type nomTableau[][];
```

- Création à l'aide de l'instruction « new »

```
nomTableau = new type [nligne][ncolonne]
```

- Exemple

```
int [][] tab;  
tab = new int [5][3];
```

Les tableaux

■ Tableaux multidimensionnels

● Déclaration

- Type [][] nomTableau;
- Type nomTableau[][];

● Création du tableau avec « new »

- nomTableau = new type [m][n];

● Exemple

```
int [][] tab;  
tab = new int [5][3];
```

Les tableaux

■ Exemples d'initialisation de tableau à la création

- Tableau à une dimension

```
int []tab = {1, 2, 3, 4, 5};
```

- Tableau à deux dimensions

```
int [][] matrice = {  
    { 1, 2, 3, 4, 5},  
    {6,7,8,9,10},  
    {11, 12, 13, 14, 15}  
}
```

Les structures de Contrôle

■ La structure alternative

```
if (condition)
  action1 ;
else
  action2;
```

```
if (i==0)
  k = k+1;
else {
  j++;
  k=j;
}
```

■ Le choix multiple

```
switch ( expression) {
  case valeur1:  action1; break;
  case valeur2: action2: break;
  default: ....
}
```

```
switch (choix){
  case 2:  k=1; break;
  case 3: k=3: break;
  default: k = 6;
}
```


Les structures de Contrôle

■ La boucle pour

```
for (expr1; expr2; expr3)
    action;
```

- **expr1**: conditions de départ (plusieurs initialisations sont possibles avec un même type)
- **expr2**: la boucle est effectuée tant que cette condition est vérifiée
- **expr3**: actions à faire à chaque fin de boucle (plusieurs instructions possibles séparées par des virgules)

```
for (int i=0, j=10; i!=j; i++,j++){
    i++;
    j--;
}
```

```
int i;
for (i=0; i<10; i++) {
    System.out.println(i);
}
```

Les structures de Contrôle

■ La boucle pour dans le cas du parcours d'une collection

```
for (type variable : tableau | collection) {  
}
```

```
int tab[] = {1,2,3,4,5,6,7,8,9};  
for (int i:tab)  
    System.out.println(i);
```

Les structures de Contrôle

■ Les boucles «tantque»

```
while (condition)
    action;
```

```
int i=2, j=10;
while (i<j)
{
    i++;
    j++;
}
```

```
do
    action;
while (condition)
```

```
int i=2, j=10;
do
    i++;
    j++;
while (i<j);
```

Les structures de Contrôle

■ Boucle pour (Parcours d'une collection)

```
for (type variable : tableau | collection) {  
}
```

● Exemple

```
int tab[] = {1,2,3,4,5,6,7,8,9};  
for (int i:tab)  
    System.out.println(i);
```

Notion de classes et d'objet

1. Classe et Objet
2. Constructeur par défaut d'une classe
3. Affectation de référence d'un objet
4. Constructeur par copie
5. La clause this
6. Attributs et méthode de classes
7. Finalisation de type, classe et méthode
8. Classes enveloppes

Classes et Objets

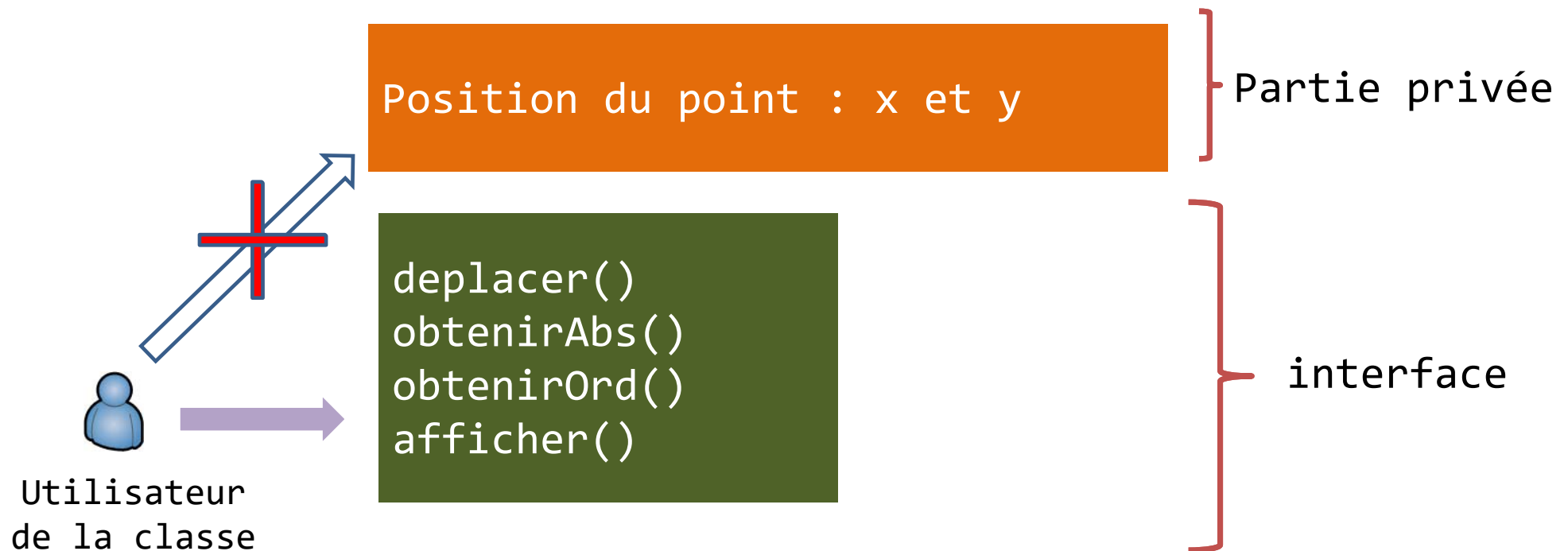
■ Définition d'une classe et d'un objet (1)

- Une classe est un modèle à partir duquel des objets peuvent être créés
- Les objets ou instances d'une même classe ont même structure et le même comportement.
- Une classe dispose d'une partie privée et d'une interface d'accès définie par une ensemble de fonctions (méthodes).
- Un objet est utilisé selon le principe de l'encapsulation (l'utilisateur a accès à l'objet uniquement via son interface)
- Deux instances d'une classe ne diffèrent que par la valeur de leurs données privées.

Classes et Objets

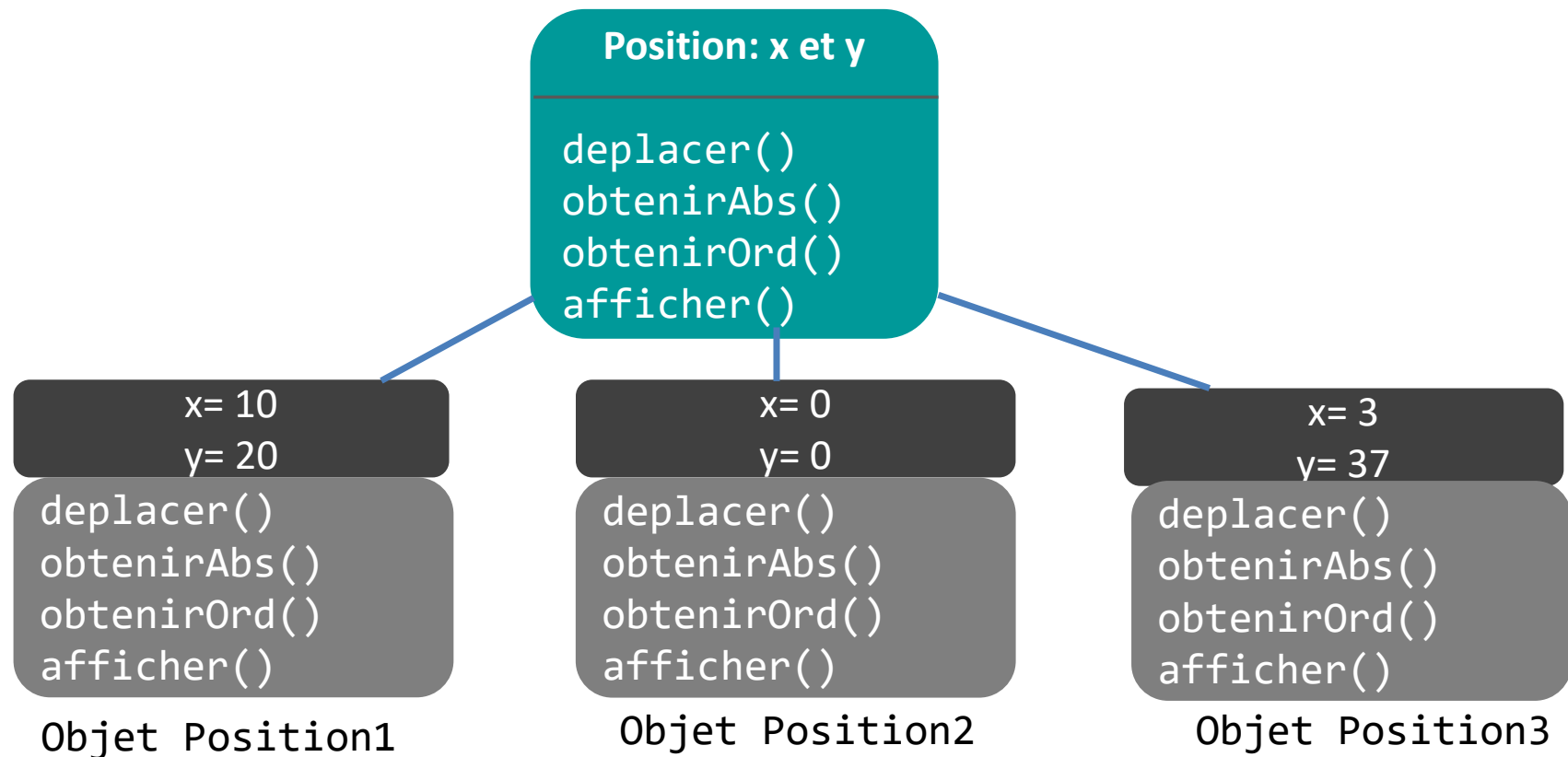
■ Définition d'une classe et d'un objet (2)

Exemple d'une classe Point



Classes et Objets

■ Définition d'une classe et d'un objet (3)



Classes et Objets

■ Déclaration de base d'une classe

```
class «NomClass» {  
    «variables d'instances ou attributs»  
    [«constructeurs»]  
    «méthodes»  
    [ public static void main(String args[]) {  
        «instruction»  
    } ]  
}
```

Classes et Objets

■ Déclaration de base d'une classe (2)

● Exemple

```
class Point {  
    int x,y;  
  
    void deplacer(int X, int Y) {x=X;y=Y;};  
    int obtenirAbs( ){return x;};  
    int obtenirOrd(){return y;};  
    void afficher(){  
        System.out.println("abscisse =" +x+"ordonnee= "+y);  
    };  
    public static void main(String[] args) {  
    }
```

Classes et Objets

■ Constructeur d'une classe

- C'est une méthode particulière de la classe appelée implicitement lors de la déclaration d'un objet de la classe
- Elle porte le nom de la classe
- Elle permet d'initialiser les attributs (variables internes ou d'état de l'objet) à créer à partir de ses paramètres ou de valeur par défaut
- Elle ne retourne aucune valeur
- Une classe peut avoir plusieurs constructeurs qui se différencient par le nombre d'argument et le type des arguments

Classes et Objets

■ Création d'un objet et invocation d'une méthode

```
nomClasse objet = new nomClasse([parametre])  
objet.methode([parametre])
```



```
Point pt1 = new Point ();  
Point pt2 = new Point(10,20);  
pt1.deplacer(10,20);
```

constructeur

■ Liste variable de paramètres d'une méthode (1)

```
«type» methode(type... params)
```

- Dans la méthode, la variable *params* représente un tableau contenant les paramètres

Classes et Objets

■ Liste variable de paramètres d'une méthode (2)

```
class A {  
    void m(int... params) {  
        for(int j:params)  
            System.out.println(j);  
    }  
  
    Public static void main(args[]){  
        A a = new A();  
        a.m(100,200,300,400);  
        a.m(2,6);  
    }  
}
```

Classes et Objets

■ Atelier 1

L'objectif de cet atelier est de créer et exécuter une classe en ligne de commande. Le programme java sera saisi dans un éditeur de texte (notepad, sublime, ou autre).

Travail à faire

- 1) Compléter la classe Point avec deux constructeurs de l'exemple: un constructeur sans paramètre et un constructeur avec deux paramètres. Dans le premier cas, les attributs internes (x et y) seront initialisés à 0. Dans le deuxième cas, les attributs seront initialisés avec les paramètres du constructeur
- 2) Créer un objet Point dans la méthode main() et appeler la méthode afficher()
- 3) Renseigner la variable d'environnement PATH avec le répertoire d'installation du compilateur Java (javac.exe) et de la machine virtuelle Java (java.exe)
- 4) Compiler et lancer la classe créée

Classes et Objets

■ Constructeur par défaut

- Constructeur sans paramètre

■ Constructeurs par défaut par défaut

- Constructeur par défaut généré par le constructeur lorsqu'aucun constructeur n'est spécifié. Dans ce cas, les attributs sont initialisés de la manière suivante:
 - ↳ Les entiers sont initialisés à 0
 - ↳ Les réels à 0.0
 - ↳ Les booléens à false
 - ↳ Les objets à Null
- Ce constructeur n'existe pas dès lorsqu'un constructeur existe

Classes et Objets

■ Constructeur par défaut, par défaut par défaut

```
public class Point {  
    int x,y;  
  
    void afficher(){  
        System.out.printf("%d %d", x,y);  
    }  
    public static void main(String[] args) {  
        Point pt = new Point ();  
        pt.afficher();  
    }  
}
```

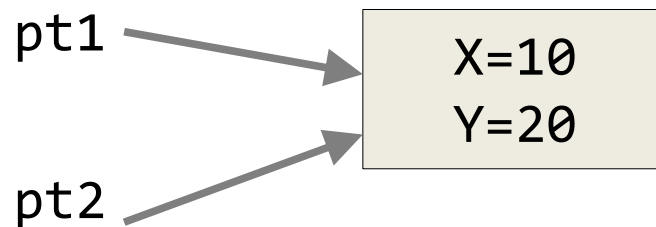
Il n'y a aucun constructeur. Donc un constructeur par défaut par défaut sera introduit

Classes et Objets

■ Affectation de référence d'objet

```
Point pt1 = new Point (10,20);  
Point pt2 = pt1;
```

- Les deux références pt1 et pt2 pointent sur le même contenu.
- Le contenu peut être alors accédé par l'une ou l'autre des deux références



Exercice

Créer un point p1. Déclarer un point p2 et affecter p1 dans p2 . Appeler la méthode afficher() sur p1 et p2
appeler la méthode déplacer() sur p2, puis, appeler la méthode afficher sur p1 et p2

Classes et Objets

■ Affectation de référence d'objet

```
Point pt1 = new Point (10,20);  
Point pt2 = pt1;
```

Classes et Objets

■ Création d'un objet à partir des attributs d'un autre objet

- Paramétrage du constructeur de la classe par la classe elle-même

```
public class Point {  
    int x,y;  
    Point(Point p){ x=p.x; y=p.y}  
    .....  
    public static void main(String[] args) {  
        Point p1 = new Point(10,20);  
        Point p2 = new Point(p1);  
    }  
}
```

- Copie d'un objet par clonage (pas étudié dans le cadre de ce cours)

Classes et Objets

■ Référence à l'objet courant: la clause *this* (1)

- La clause «*this*» fait référence à l'objet courant auquel s'applique la méthode où elle est utilisée
- Suivi d'un point, elle permet d'accéder à une méthode ou un attribut de l'instance de l'objet courant

```
Point (int x, int y) {  
    this.x = x ;  
    this.y = y ;  
}
```

Classes et Objets

■ Référence à l'objet courant: la clause `this` (2)

- Utilisée comme un appel à une méthode (`this([paramètres])`), elle permet d'appeler un constructeur de la classe courante
 - Dans ce cas «`this`» ne peut être utilisée qu'en première instruction dans un corps de constructeur

```
Point (int x, int y) {  
    this.x = x ;  
    this.y = y ;  
}
```

```
Point(int x, int y, int coul){  
    this(x,y);  
    this.coul = coul;  
}
```

Classes et Objets

■ Attribut de classe

- Elle est indépendante d'un objet
- Il existe un seul exemplaire de l'attribut pour tous les objets
- Tous les objets de la classe référencent l'unique exemplaire de l'attribut de classe
- Déclaration

```
static type nomAttribut
```

```
class Point {  
    static int cpt;  
};
```

Classes et Objets

■ Accès à un attribut de classe

- Elle peut se faire en utilisant l'objet

```
Point unPoint = new Point();  
unPoint.cpt = 1;
```

- Elle peut se faire directement par le nom de la classe

```
Point.cpt = 1;
```

Exercice

Compléter la classe Point afin de compter le nombre d'objet Point créés

Classes et Objets

■ Méthode de classe

- Une méthode de classe ne dépend pas d'un objet
- Elle peut être accéder directement avec le nom de la classe (elle est accessible aussi bien via un objet)
- Dans une méthode de classe, on ne peut accéder qu'aux attributs ou méthodes de classes.
- Déclaration et utilisation

```
static type nomMethode(.....)
```

```
static int nombre(); int nb = Point.nombrePoint();  
Point p = new Point(); p.nombrePoint();
```


Classes et Objets

■ La clause «final»

- final «Variable de type de base» → est initialisée seule fois

```
final int i=0;          final int j; j=10;
```

- final « référence d'objet » → La référence créée ne peut pas être modifiée

```
final Point pt = new Point ();
```

- final « Méthode d'instance » → La méthode ne peut plus être redéfinie

```
final void m();
```

- final «classe » → La classe ne peut pas avoir de sous classe

```
final class C { }
```

- Un attribut de classe «final » doit être initialisé à la déclaration ou dans tous les constructeurs de la classe

Classes et Objets

■ Les classes enveloppes (wrappers)

- Une classe enveloppe encapsule les données d'un type primitif

Type primitif	Classe enveloppe
int	Integer
short	Short
long	Long
boolean	Boolean
char	Character
byte	Byte
float	Float
double	Double
void	Void

Classes et Objets

■ Autoboxing

- Il permet de transformer un type primitif en une instance de la classe enveloppe

```
Integer i = 3;    // int -> Integer
```

```
Long l = 3L;      // long -> Long
```

```
Long l = 3;    // erreur, int -> Integer -> Long
```

■ Auto-unboxing

- Il permet de transformer une instance d'une classe enveloppe dans le type primitif correspondant

```
Integer i = new Integer(3);
```

```
int x = i; // Integer -> int
```

Classes et Objets

■ Conversions de chaîne de caractères en type primitif

- Hormise la classe Character, chaque classe enveloppe dispose d'une méthode de classe pour convertir une chaîne dans le type primitif correspondant.

```
parseX(String s) throws NumberFormatException  
X est le nom du type primitif
```

```
i= Integer.parseInt("500");  
Double d = Double.parseDouble("45.5");
```

Classes et Objets

■ Atelier 2

Une pile est une structure de données représentant un ensemble (séquence) d'éléments qui fonctionne selon le principe de «dernier arrivé, premier sorti » ou LIFO (Last in, First Out). Elle fonctionne donc comme une pile d'assiettes. Le dernier élément ajouté au sommet de la pile sera le premier élément à être récupéré. Les opérateurs nécessaires sur une pile sont:

- Tester si la pile est vide
- Tester si la pile est pleine
- Empiler un élément x au sommet de la pile
- Dépiler l'élément x qui se trouve au sommet de la pile
- Afficher le contenu de la pile

Dans cet atelier, on décide de représenter une pile d'entier par un tableau d'entier. Dans ce cas, sommet représente l'indice de l'élément au sommet de la pile. Donc avant d'ajouter un élément dans la pile, l'attribut sommet sera d'abord incrémenté. De même après avoir récupéré l'élément de sommet de la pile l'attribut sommet sera décrémenté.

Ecrire et tester la classe Pile (un IDE peut être utilisé)

Classes et Objets

■ Atelier 3

Dans ce atelier on souhaite réaliser quelques opérations sur une matrice (tableau à deux dimensions) de N lignes et M colonnes. La matrice sera représentée par une classe Matrice qui va encapsuler un tableau à deux dimensions, ainsi que les différentes dimensions.

Un objet matrice sera créé, soit à partir d'un tableau primitif à deux dimension ou à partir d'un objet matrice existant

Ecrire et tester la classe matrice dotée des méthodes suivantes:

- Constructeurs
- Addition de deux matrices (le résultat est la matrice courante)
- Le produit de deux matrices (le résultat est la matrice courante)

L'héritage

1. Définition et déclaration
2. Référence à la classe de base: clause super
3. Redéfinition et surcharge de méthode
4. Résolution de lien statique et dynamique
5. Polymorphisme

Héritage

■ Héritage

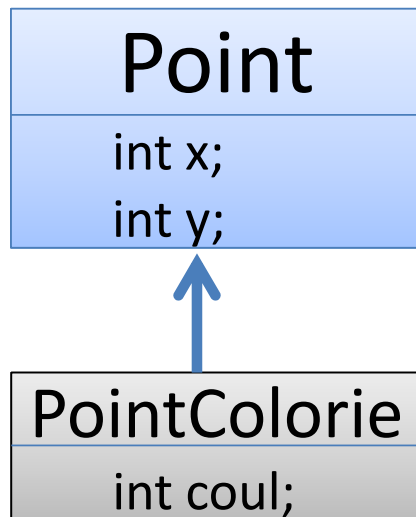
- Mécanisme qui permet la réutilisation des attributs et méthode d'une classe existante pour construire une nouvelle classe.
- Dans une hiérarchie d'héritage, les classes sont liées par la relation « **est un** » ou « **est une sorte de** ».
- La classe qui hérite est appelée sous classe ou classe dérivée ou classe fille.
- La classe héritée est dite classe de base ou classe mère ou super classe.

■ Déclaration

```
class «NomClasseDerivee» extends «NomClasseBase» {  
  ...  
}
```


Héritage

■ Exemple



```
class PointColorie extends Point {
    int coul;

    PointColorie(int coul){
        ...
    }
    PointColorie(int x, int y, int coul){
        ...
    }
}
```

Héritage

■ Référence à la classe de base: La clause super

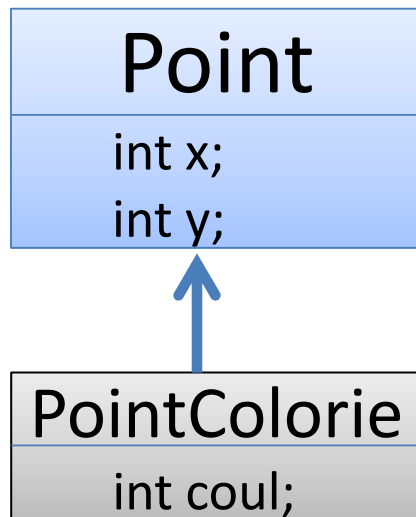
- La clause « super » fait référence à la classe de base de l'objet.
- Suivie d'un point, elle permet d'accéder à une méthode ou un attribut de l'instance courante de la classe de base.

`super.cpt = 6`

- Utilisée comme une méthode, elle permet d'appeler un constructeur de la classe de base de l'objet en cours.
 - ▶ Dans ce cas, la clause « super » ne peut être utilisé qu'en première instruction dans un corps de constructeur

Héritage

■ Exemple



```
class PointColorie extends Point {
    int coul;

    PointColorie(int coul){...}
    PointColorie(int x, int y, int coul){
        ...}
}
```

Héritage

■ Exemple

```
class PointColorie extends Point {
    int coul;

    PointColorie(int coul){
        super(); // appel au constructeur sans parametre de la classe Point
        this.coul = coul;
    }
    PointColorie(int x, int y, int coul){
        super(x,y); // appel au constructeur de la classe Point ayant 2 paramètres
    }
    void afficher(){
        super.afficher();System.out.println("Couleur = "+coul);
    };
}
```

Héritage

■ Redéfinition (masquage) d'une méthode (overriding)

- La redéfinition d'une méthode héritée doit impérativement conserver la signature de la méthode parent (type et nombre de paramètres, valeur de retour et exceptions propagées doivent être identiques).
- Dans l'exemple précédent la méthode *void afficher()* de la classe *PointColorie* surcharge la méthode *void afficher()* de la classe *Point*
- A partir de java 5.0, lorsqu'une méthode renvoie un objet d'une classe donnée C, sa redéfinition peut renvoyer un objet d'une autre classe D à condition que la classe D hérite de C (idem pour les exceptions propagées).



Héritage

■ surcharge d'une méthode (overloading)

- La surcharge d'un constructeur permet de définir le constructeur plusieurs fois avec le même nom mais avec des arguments différents.

```
class Poin{  
    int x, y;  
    Point(){...}  
    Point(int x, int y){...}  
}
```

- La surcharge d'une méthode ordinaire consiste à définir la méthode plusieurs fois avec le même nom mais avec des arguments différents en nombre et en type, et avec le type de retour éventuellement différent.
- Une méthode héritée peut être surchargée (c'est normal !).

Héritage

■ Héritage et sous type

- Lorsqu'une classe B hérite d'une autre classe C, le type de la classe dérivée devient un sous-type de celui de classe de base.
- Une instance de la classe dérivée peut être utilisée partout où une instance de la classe de base est attendue
- On peut affecter un objet de type sous-classe à une variable de type super-classe

```
PointColorie pc = new PointColorie(10,20,300);
```

```
Point p = pc;
```

```
Pc = p;
```

Héritage


■ Choix de la méthode à exécuter en cas de redéfinition (1)

- Lorsqu'une instance de sous-classe B est substituable à une instance de la classe mère A, que se passe-t-il lorsque B redéfinit une méthode de A ?
- Résolution statique des liens
 - ▶ C'est le type de la variable qui détermine la méthode à exécuter

```
class A {  
void afficher(){System.out.println("A");}  
};
```

```
class B extends A {  
void afficher(){System.out.println("B");}  
};
```

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



Héritage

■ Choix de la méthode à exécuter en cas de redéfinition (2)

● Résolution dynamique des liens

- ▶ Le choix de la méthode à exécuter est fonction de la nature réelle de l'instance contenue dans la variable
- ▶ Java met systématiquement en œuvre la résolution dynamique des liens

```
class A {  
void afficher(){System.out.println("A");}  
};
```

```
class B extends A {  
void afficher(){System.out.println("B");}  
};
```

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



Héritage

■ La super-classe Object (1)

- La classe «Object» est la super-classe commune à toutes les classes en Java
- Toute classe Java, prédéfinie ou personnelle hérite de la super classe *Object*
- Il est donc possible d'affecter une instance de n'importe quelle classe à une variable de type Object

```
Object pco = new Point(10,20,300);
```

Héritage

■ La super-classe Object (2)

```
public class Object {  
    public String toString() {...}  
    protected Object clone() throws CloneNotSupportedException{  
        ...}  
    public boolean equals(Object obj) {...}  
    .....  
}
```

- ▶ Par défaut la méthode *toString* renvoie une chaîne représentant l'adresse mémoire de l'objet. Cette méthode est appelée par `System.out.print()` lorsqu'on veut afficher un objet.
- ▶ La méthode *clone()* permet de réaliser le clonage d'un objet
- ▶ La méthode *equals()* permet de comparer deux objets

Héritage

■ La super-classe Object (4)

● Exemple 1

```
Point p = new Point (20,40);  
System.out.println(p); → affiche: Point@7852e922
```

● Exemple 2 (Méthode redéfinie *equals* dans la classe Point)

```
public boolean equals (Object op) {  
    Point px = (Point)op;  
    if (x == px.x && y == px.y)  
        return true;  
    else  
        return false;  
};
```

Héritage

■ Le polymorphisme

- C'est une technique dans laquelle une même méthode s'exécute différemment selon la donnée à laquelle elle s'applique
- Il est mis en œuvre par deux techniques fondamentales:
 - ▶ Le mécanisme d'héritage
 - ▶ La résolution dynamique de lien
- Il existe deux types de polymorphisme
 - ▶ Le polymorphisme des traitements
 - ▶ Le polymorphismes des données

Héritage

■ Le polymorphisme des traitements (polymorphisme ad'hoc)

- Il s'agit du mécanisme de surcharge des méthodes
- Le même identificateur est utilisé pour désigner des séquences d'instructions différentes

■ Le polymorphisme des données (1)

- Polymorphisme paramétrique
 - ▶ Il est connu sous le nom de généricité où une classe est paramétrée par un type T
 - ▶ Le même code est écrit pour le type T est applicable à n'importe quelle classe compatible
 - ▶ Il est le plus souvent utilisé pour les collections génériques

Héritage

■ Le polymorphisme des données (1)

● Polymorphisme d'inclusion

- ▶ Les instances d'une sous-classe sont substituables aux instances des classes de son ascendance, en argument d'une méthode ou lors d'affectations, tout en gardant leurs natures/propriétés propres.
- ▶ Il permet de substituer à l'exécution un appel de méthode par une autre en fonction de la classe de l'objet sur lequel s'applique la méthode
- ▶ Exemple

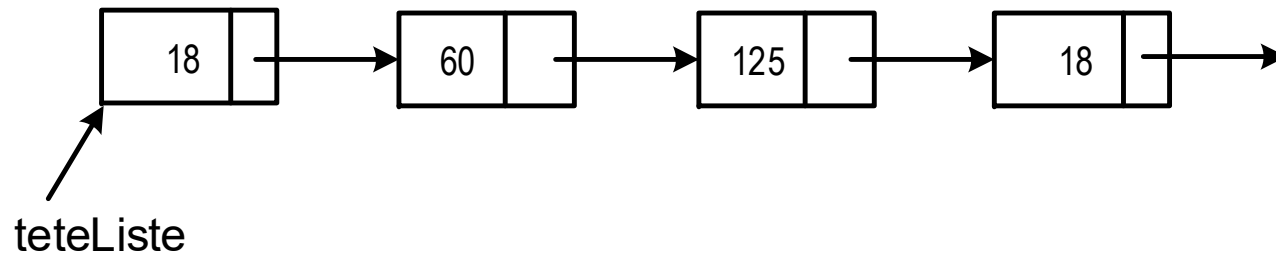
```
Point pc = new PointColorie(10,20,300);  
pc.afficher();
```

A l'exécution, c'est la méthode *afficher()* définie au niveau de la classe *PointColorie* qui sera appelée et non la méthode *afficher()* de la classe *Point*

Héritage

■ Atelier 4 (1)

Dans ce atelier, on se propose d'implémenter une classe Pile d'entier par réutilisation d'une liste linéaire chaînée d'entiers. Une liste chaînée est une structure de données telle que la connaissance d'un élément permet de connaître l'élément suivant. En effet, l'élément courant contient, en plus de la donnée, la référence vers l'élément suivant. Le dernier élément de la liste contient une référence nulle pour l'élément suivant.



On peut insérer un élément en début de liste, en fin de liste, à n'importe quelle position de la liste.

Chaque élément de la liste contient deux champs: un nombre et la référence vers l'élément suivant. Un élément sera donc représenté par une classe ayant les deux attributs et un constructeur comme seule méthode

Héritage

■ Atelier 4 (2)

L'élément de la liste est implémentée par la classe Cellule

```
class Cellule {  
    int info;  
    Cellule suiv;  
    Cellule(int info, Cellule suiv){  
        this.info=info;  
        this.suiv=suiv;  
    };  
};
```

- 1) Une liste est définie par la donnée de son premier élément. Ecrire la classe Liste avec les méthodes d'insertion et tête de liste et d'insertion en fin de liste
- 2) Ecrire une classe Pile qui hérite de la classe Liste.

Classe abstraite et interface

1. Définition et déclaration d'une classe abstraite
2. Définition et déclaration d'une interface
3. Implémentation d'une interface

Classe abstraite et interface

■ Classe abstraite (1)

- Une classe abstraite a pour objectif de «factoriser» les attributs et le comportement d'un ensemble de classes qui deviennent des sous types d'un même type
- Ses méthodes sont redéfinies dans les classes dérivées
- Déclaration

```
abstract class « Nom-Classe»  
    {  
        abstract methode1 ();  
        void methode2(){};  
    }
```

Classe abstraite et interface

■ Classe abstraite (2)

- Une classe dérivée d'une classe abstraite doit impérativement redéfinir toutes les méthodes abstraites, à moins qu'elle soit déclarée elle-même abstraite,
- Une classe dérivée d'une classe non abstraite peut être abstraite
- Exemple

```
abstract class Animal {  
    abstract void crier();  
}
```

```
class Chien extends Animal {  
    void crier(){...};  
}
```

Classe abstraite et interface

■ Interface

- Une interface est une classe abstraite, sans constructeur
- Toutes les variables internes (attributs) sont des constantes implicitement **static** et **final**
- Toutes les méthodes sont implicitement **abstraites et publiques**
- Une classe qui implémente une interface donne un corps à toutes les méthodes de l'interface
- Une interface peut étendre une ou plusieurs interfaces
- Une classe peut implémenter une ou plusieurs interfaces

Classe abstraite et interface

■ Interface

● Définition

```
interface «nom-interface» [extends  
«Liste-interface-base»] {  
}
```

● Implémentation

```
class «nom-classe» implements  
«nom-interface» {  
}
```

Classe abstraite et interface

■ Interface

● Exemple

```
interface formeGeometrique {  
    float surface();  
    float périmètre();  
}
```

● Implémentation

```
class Rectangle implements formeGeometrique {  
    float surface() {};  
    float surface() {};  
}
```

Classe abstraite et interface

■ Interface

- A partir de Java 8

- ▶ Il est possible de définir des classes statiques dans un interface
- ▶ Il est possible de fournir un code par défaut à une méthode via la clause **default**

```
interface formeGeometrique {  
    int PAS_ROTATION = 30;  
    static int getPasRotation() {return PAS_ROTATION;};  
    default void setCouleur(){};  
};
```


Classe abstraite et interface

■ Utilité des interface

- Une classe abstraite sert à factoriser des état et comportement similaires à un ensemble de classe apparentées.
- Une interface sert à définir un contrat de services avec des classes utilisatrices, même si ces classes n'ont pas de lien de similarité entre elles.
- L'interface dissimule une implémentation pour la classe appelante , ce qui permet de modifier l'implémentation sans modifier la classe.
- Le plus souvent, elle sert d'intermédiaire entre une classe utilisatrice et une implémentation
- La notion d'interface est beaucoup utilisée dans le cadre du développement des API Java

■ Atelier 5

L'objectif de cet atelier est de mettre en œuvre les concepts de polymorphisme, de classe abstraite et d'interface.

On suppose qu'une forme géométrique est une figure dont on peut calculer la surface et le périmètre. Une forme géométrique est aussi par sa position (objet de la classe *Point*) dans le plan. Le triangle, le cercle, le carré, le rectangle sont des exemples de formes géométrique.

- 1) Ecrire la classe abstraite nommé *formeGeo* représentant les formes géométrique. Elle sera doté des méthodes abstraites *float surface()* et *float périmètre()* qui renvoient respectivement la surface et le périmètre de la forme géométrique.
- 2) Ecrire les classes *Rectangle* et *cercle* qui redéfinissent les méthodes précédentes
- 3) Un domaine est un ensemble de forme géométrique. La surface d'un domaine est la somme des surfaces des forme géométrique qui le composent. En supposant que le domaine est un tableau de forme géométriques, écrire la classe domaine dotée de la méthode *float surface*.
- 4) Reprendre les question 1,2 et 3 en faisant de la classe abstraite *formeGeo*, une interface.

Paquetage (Package)

1. Définition et principe de nommage des packages
2. Accessibilité des classes, attributs et méthodes
3. Création et importation d'un paquetage
4. Compilation, exécution et Paquetage
5. Variable d'environnement PATH et CLASSPATH

Les paquetage (package)

■ Définition

- Un package est un regroupement de classes ayant un centre d'intérêt commun
- Il est implémenté par un répertoire
- Un package est aussi un espace de nom pour éviter des conflits de nom

■ Principe de nommage

- Un package porte le nom du répertoire qui le contient, les éléments constitutifs du répertoire étant séparés par « . »
- Il est recommandé d'utiliser comme base de l'arborescence de package le nom de domaine inversé de l'organisation

- Exemple

- ▶ `ci.inphb.larima.jse`

- `ci/inphb/larima/jse` (Linux)

- `ci\inphb\larima\jse` (Windows)

Les paquetage (package)

■ Accessibilité des variables et méthodes

- L'accès à un attribut ou à une méthode d'une classe par une autre est lié à la structuration en package.

	Variables, méthodes d'une classe			
	Public	Protected	Par défaut	Private
Dans la même classe	OUI	OUI	OUI	OUI
Une autre classe du même package	OUI	OUI	OUI	NON
Une sous classe du même package	OUI	OUI	OUI	NON
Une sous classe extérieur au package	OUI	OUI	NON	NON
Une classe extérieure au package	OUI	NON	NON	NON

Les paquetage (package)

■ Accessibilité des attributs et méthodes

package ci.inphb.larima

```
class Base {  
    public int pb;  
    protected int pt;  
    private int pr;  
    int df;  
};  
  
class C {  
    pb, pt, df  
    pr  
}  
  
class D extends Base{  
    pb, pt, df  
    prive  
};
```

package ci.inphb.esi

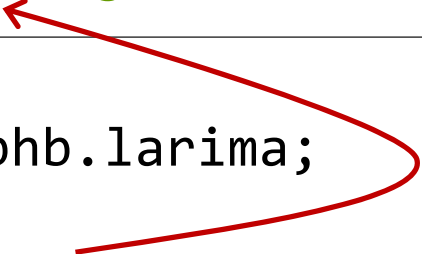
```
class CE {  
    pb  
    pt, df, pr  
}  
  
class DE extends Base{  
    pb, pt  
    df, pr  
    pt ne doit pas être accédé avec un  
    Objet de la classe Base,  
};
```

Les paquetage (package)

■ Remarque

Dans le cas où il existe une classe publique dans le package, le fichier contenant le paquetage doit impérativement porter le nom de cette classe publique.

Fenetre.java



```
package ci.inphb.larima;  
  
public class Fenetre {  
  
}
```

Les paquetage (package)

■ Création d'un package

- Elle se fait dans le programme, en première instruction à l'aide de la commande suivante :

```
package nom_paquetage ;
```

- Exemple

```
package ci.inphb.larima;  
  
class Fenetre {  
  
}
```


Les paquetage (package)

■ Importation de package (1)

- A l'instanciation d'une classe, la JVM recherche la classe en question et la charge en mémoire. Plusieurs méthodes d'accès sont utilisées
- Accès à une classe par le nom complètement qualifié

```
ci.inphb.larima.Point.afficher();
```

- Accès par importation d'une classe particulière

```
import ci.inphb.larima.Point;
```

- Accès par importation de toutes les classes du package

```
import ci.inphb.larima.*;
```

Les paquetage (package)

■ Importation de package et sous package

- Un package peut avoir des sous packages
`ci.inphb.larima` est un sous package de `ci.inphb`
- L'importation des classes d'un package n'importe pas les classes des sous packages. Les 2 importations suivantes n'importent pas les mêmes classes

```
import ci.inphb.larima.*;  
import ci.inphb.*;
```

Les paquetage (package)

■ importation de membres statique d'un paquetage (1)

- Il est possible d'importer uniquement des variables ou méthodes statiques d'une classe. Dans ce cas, elles sont accédées sans les préfixer du nom du package.
- Exemple 1: sans importation des membres statiques

```
class SansStatic {  
    public static void main(String args[]) {  
        double racine = Math.sqrt(5.0);  
        double tangente = Math.tan(30);  
        System.out.println("Racine carree = "+ racine);  
        System.out.println("Tangente de 30="+ tangente);  
    }  
}
```

Les paquetage (package)

■ importation de membres statique d'un paquetage (2)

- Exemple 1: avec importation des membres statiques

```
import static java.lang.Math.*;
class AvecStatic {
    public static void main(String args[]) {
        double racine = sqrt(5.0);
        double tangente = tan(30);
        System.out.println("Racine carree = "+ racine);
        System.out.println("Tangente de 30="+ tangente);
    }
}
```

- Importer d'une seule variable statique

```
import static java.lang.Math.PI;
```

Les paquetage (package)

■ Les variable d'environnement PATH et CLASSPATH

- La variable PATH contient les répertoires dont les programmes exécutables peuvent être lancés à partir de n'importe quel autre répertoire.
- La variable CLASSPATH contient les répertoires à partir duquel java recherche les packages.
- Ajout d'un répertoire dans PATH et CLASSPATH (Windows)

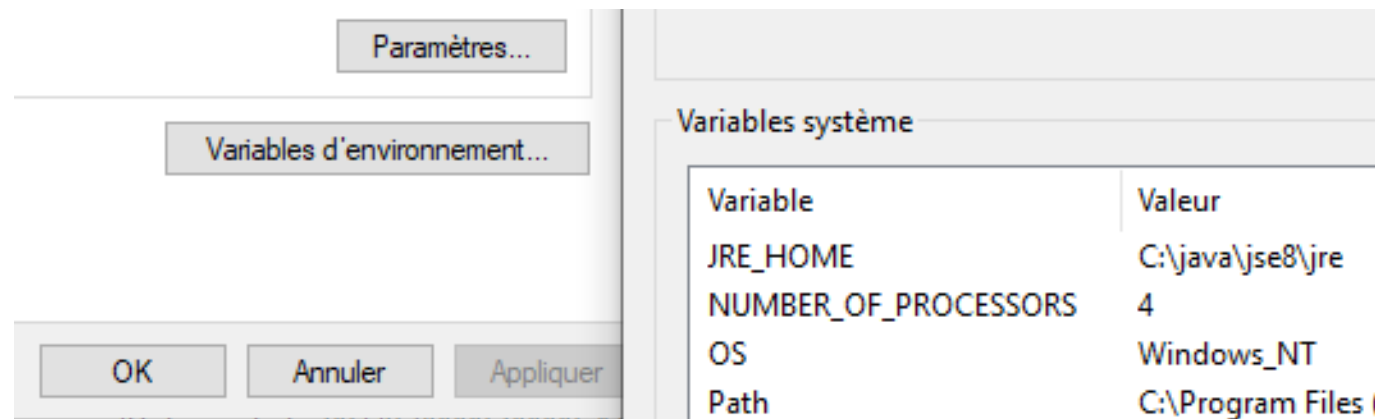
```
PATH=%PATH%;Repertoire  
Set CLASSPATH=%CLASSPATH%;Repertoire
```

```
PATH=%PATH%;C:\java\packages  
set CLASSPATH=%CLASSPATH%;C:\java\packages;.
```

Les paquetage (package)

■ Les variable d'environnement PATH et CLASSPATH

- Les deux variables peuvent être renseignées dans les paramètres systèmes avancés de Windows



Les paquetage (package)

■ Compilation en ligne de commande d'un programme Java (1)

```
javac      [-d    repertoire_racine_Package]  programme.java
```

- L'option `-d` crée et place automatiquement les classes du paquetage dans le répertoire portant son nom à partir du répertoire racine indiqué.

```
javac -d c:\java\packages Fenetre.java
```

- Après compilation, les bytecode des classes contenues dans `Fenetre.java` seront mises dans le package indiqué le fichier à partir du répertoire `c:\java\packages`

Les paquetage (package)

■ Compilation en ligne de commande d'un programme Java (2)

- Pendant la compilation ou l'exécution Java recherche un paquetage dans les répertoires spécifiés dans la variable d'environnement «*classpath*»
- Ces répertoires de recherche peuvent être indiqués directement en ligne de commande par l'option «*-cp (classpath)* » du compilateur ou de la JVM

```
javac -cp «rep1»;«rep2» programme.java
```

```
javac -cp c:\java\packages Fenetre.java
```

```
java -cp c:\Java\packages;. testFenetre.java
```


Les paquetage (package)

■ Atelier 5 (1)

Ce atelier sera réalisé dans un premier temps en ligne de commande, puis dans un IDE
Soit D:\Java\TP, votre répertoire de travail dans lequel vont se trouver tous vos programmes Java.

On se propose d'utiliser des classes se trouvant dans le package `ci.inphb.larima` relativement au répertoire D:\java\packages (workspace de l'installation de l'IDE Eclipse).

Le fichier `Point.java` contient le code de la classe `Point` et se trouve dans le répertoire D:\java\TP. Une fois ce fichier compilé, les classes *Point.class* se trouvera dans le répertoire: D:\java\packages\ci\inphb\larima

Le fichier `Fenetre.java` contient le code de la classe `Fenetre`, y compris la fonction principale `main()`. Il se trouve dans le répertoire D:\java\TP. Une fois compilé, la classe *Fenetre.class* se trouvera dans le répertoire courant. Cette classe fait appelle à la classe *Point*.

Les paquetage (package)

■ Atelier 5 (2)

Travail à faire

Un objet de la classe *Point* est caractérisé par ses coordonnées (abscisse et ordonné). En plus de deux constructeurs, les méthodes de la classe *Point* sont:

- void `deplacer (int x, int y)`: modifie les coordonnées du point
- void `afficher()`: affiche les coordonnées du point

La classe Fenêtre est caractérisée par sa position qui est un objet de la classe *Point*, sa largeur et sa longueur. Les méthodes sont:

- void `deplacer (Point p)`: déplacer la fenêtre à un autre point
- void `afficher()`: afficher la position, la largeur et la longueur de la fenêtre.

En respectant les recommandations, écrire les classes *Point* et classes, compiler et tester.

Les exceptions

1. Définition
2. Hiérarchie des classes d'exception
3. Levée implicite et traitement d'une exception
4. Levée explicite et traitement d'une exception
5. Exception utilisateur
6. Propagation d'une exception

Définition

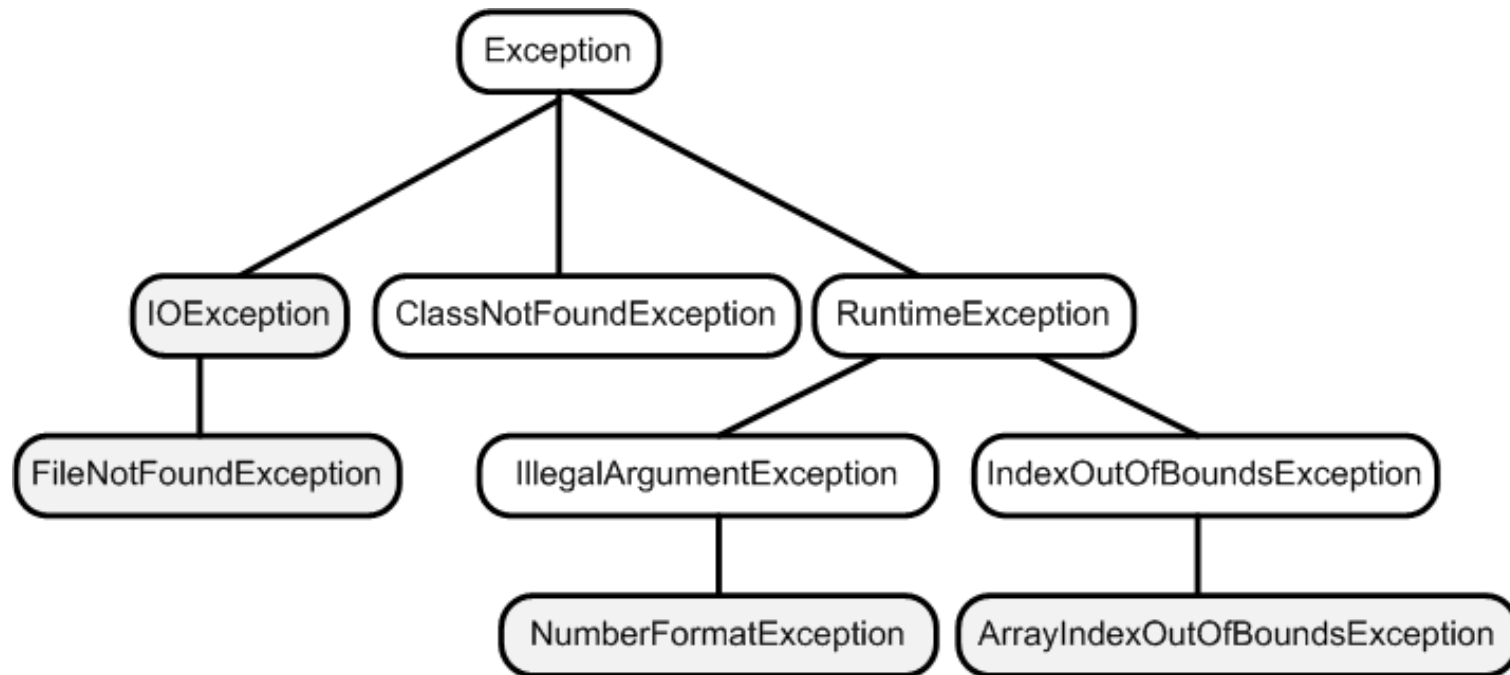
■ Une exception

- Événement exceptionnel ou erreur inattendue qui se produit lors de l'exécution d'un Programme et qui interrompt l'enchaînement normal des instructions
- Exemples de source d'exception :
 - ▶ Saisie d'une chaîne de caractère à la place d'un nombre
 - ▶ Accès à un indice hors limite dans un tableau
 - ▶ Ouverture d'un fichier qui n'existe pas
 - ▶ Accès à une imprimante manquant du papier,
 - ▶ Accès à un disque plein, manque de mémoire

Hiérarchie des classes d'exception

■ La classe Exception

- Les exceptions sont gérées par la hiérarchie de classe Exception



Levée d'une Exception

- Déclenchement (levée) implicite d'une exception (par l'interpréteur du langage)
 - Division par zéro,
 - Accès hors des bornes d'un tableau,
 - Fichier inexistant,
 - Mauvaise saisie

- Déclenchement ou levée explicite d'une exception
 - Via l'instruction *throw (e) ou throw e*, *e* = objet d'une classe d'Exception

Traitement d'une Exception

■ Mode de gestion d'une exception

- Une exception levée peut être Traitée immédiatement dans la méthode où elle se produit
- Elle peut aussi être reportée pour être traitée dans une méthode appelant la méthode où elle se produit

■ Traitement d'une exception dans la méthode courante

- Tout code susceptible de lever une exception doit être inclus dans un bloc «try-catch»

```
try { }  
catch (ClassException ce){ }
```

Traitement d'une Exception

■ forme complète de la clause try { } catch{ }

```
try {
```

```
..... •
```

```
code susceptible de lever une exception de classe:
```

```
ClassException1 ou ClassException2 ou ClassException3
```

```
..... •
```

```
}
```

```
catch(ClassException1 e){.....}
```

```
catch(ClassException2 e){.....}
```

```
..... • •
```

```
catch(ClassExceptionk e){.....}.
```

```
[Finally() {}];
```


Traitement d'une Exception

■ Exemple

```
public class TestException {  
    public TestException() {  
    }  
    public static void main(String[] args) {  
        int i=5, j=0;  
        try {  
            System.out.println(i/j); // levée d'une exception j=0  
            System.out.println("Suite du code de try !!!!");  
        } catch (ArithmeticException e) {  
            System.out.println("Exception Division par zéro !!!!!!!");  
        }  
        System.out.println("Suite du code .....");  
    } }  
}
```

Traitement d'une Exception

■ Ordre d'exécution des clauses catch relatifs à un try (1)

- Les clauses «catch» sont consultés par l'interpréteur java dans l'ordre de leur déclaration
- Les cas d'Exception spécifiques doivent donc être déclarés avant les cas d'exception généraux
- Par exemple, un cas d'exception *ArithmeticException* doit être défini avant le cas d'exception général *Exception*.

Traitement d'une Exception

■ Ordre d'exécution des clauses catch relatifs à un try (2)

● Exemple

```
try {  
    // code susceptible de lever les exceptions de type  
    // ArithmeticException et ArrayIndexOutOfBoundsException  
}  
catch (ArithmeticException e) {  
}  
catch (ArrayIndexOutOfBoundsException e){  
}  
catch (Exception e) {  
}
```

Traitement d'une Exception

■ Plusieurs classes d'Exception par clause catch

- Il est possible de spécifier plusieurs classes d'exception par clause catch

```
try {  
.....  
code susceptible de lever une exception de classe:  
ClassException1 ou ClassException2 ou ClassException3  
.....  
}  
catch(ClassException1|ClassException2|ClassException2 ex){...}
```

- Remarque

- ▶ La variable *ex* est implicitement final
- ▶ La liste des classes d'exception ne doit pas comporter une classe et ses sous classes

Traitement d'une Exception

■ La clause « finally »

- Le code contenu dans cette clause est exécutée dans tous les cas, c'est-à-dire qu'il ait levée ou pas d'exception
 - ▶ Cas de levée d'une exception suivie de son traitement
 - ▶ Cas de levée d'une exception sans traitement prévu
 - ▶ Cas de non levée d'exception

```
try { }  
[catch(ClassException e) {}]  
finally { }
```

Traitement d'une Exception

■ Atelier 1

Soient trois tableaux de N entiers: a , b et c . Le tableau c est construit à partir des tableaux a et b avec $c[i] = a[i]/b[i]$. Les tableaux a et b sont initialisés à la déclaration. Au moins un élément du tableau b est nul.

Ecrire le programme Java pour construire le tableau c , en gérant les cas d'exception. Lorsque $b[i]$ vaut 0, alors $c[i]$ reçoit 0.

Exceptions: Travaux Pratique

■ Atelier 2

Une classe dispose d'une méthode paramétrée par un tableau de chaîne de caractères. En principe chaque chaîne devrait être formée de chiffres de telle sorte que la méthode retourne la moyenne des éléments du tableau. Mais il se peut que certaines chaînes contiennent des caractères. Pour cela, on utilisera la fonction *Integer.parseInt(String s)* qui convertit une chaîne de caractère en un nombre entier, si cela est possible; Dans le cas où la conversion n'est pas possible une exception de type *NumberFormatException* est générée.

- 1) Ecrire et tester la classe Java correspondante
- 2) Transformer cette classe pour que les éléments du tableau puissent être entrés en ligne de commande.

Levée Explicite d'une Exception

■ La clause « throw »

- Cette clause permet de lever explicitement une exception

```
throw objException;
```

- Exemple

```
Scanner clavier = new Scanner(System.in);
try {
    j = clavier.nextInt();
    if (j==0)
        throw new ArithmeticException();
} catch (ArithmeticException e) {
    System.out.println("Exception!!!!!!");
}
```


Exception utilisateur

■ Principe

- Une classe d'exception peut hériter pour construire une classe « utilisateur »
- Exemple

```
public class ExceptionUtilisateur extends Exception {  
  
}
```

```
try {  
    throw new ExceptionUtilisateur();  
}  
catch(ExceptionUtilisateur eu ){  
    System.out.println(eu.getMessage());  
}
```

Propagation d'une Exception

■ Principe

- Elle consiste à ne pas traiter une exception dans une méthode « p » où elle a lieu, mais de «reporter» son traitement au niveau d'une méthode appelante « ».

```
void p() throws E1, E2 {
```

Code avec possibilité de levée des exceptions de classe E1 ou E2

```
}
```

```
void m() {  
try {  
    p(); // appel à p  
}  
catch( E1 e) { }  
catch( E2 e) { }  
}
```

Propagation d'une Exception

■ Atelier 3

Une classe contient trois tableaux de N entiers a, b et c. Cette classe dispose d'une méthode *void remplir()* qui permet de construire le tableau c à partir des tableaux a et b avec $c[i] = a[i] / b[i]$. Les tableaux a et b sont initialisés à la déclaration. Au moins un élément du tableau b est nul.

La fonction remplir est évoquée dans la méthode principale *main()*. La méthode *remplir()* ne traite pas les exceptions, mais les reporte au niveau supérieur. Ecrire le programme java en gérant les cas d'exception.

Les flux d'entrée - sortie

1. Qu'est ce qu'un flux d'Entrée – Sortie ?
2. Les flux d'octets
3. Les flux de caractères
4. La gestion des fichiers et répertoire
5. La sérialisation des objets

Gestion des flux d'entrée - sortie

■ Qu'est ce qu'un flux d'entrée-sortie ?

- Un flux est un canal de communication entre une source et une destination
- Sur ce canal les données sont écrites ou lues séquentiellement
- Un flux est associé à une source (flux d'entrée) ou à une destination (flux de sortie)
- La source et la destination peut être:
 - ▶ Un fichier sur le disque dur de l'ordinateur
 - ▶ Une zone de mémoire (tampon mémoire)
 - ▶ Une chaîne de caractères
 - ▶ Une connexion réseau
 - ▶ Un autre flux.

Gestion des flux d'entrée - sortie

■ Les classes de base de gestion de flux

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

■ Les flux d'octets

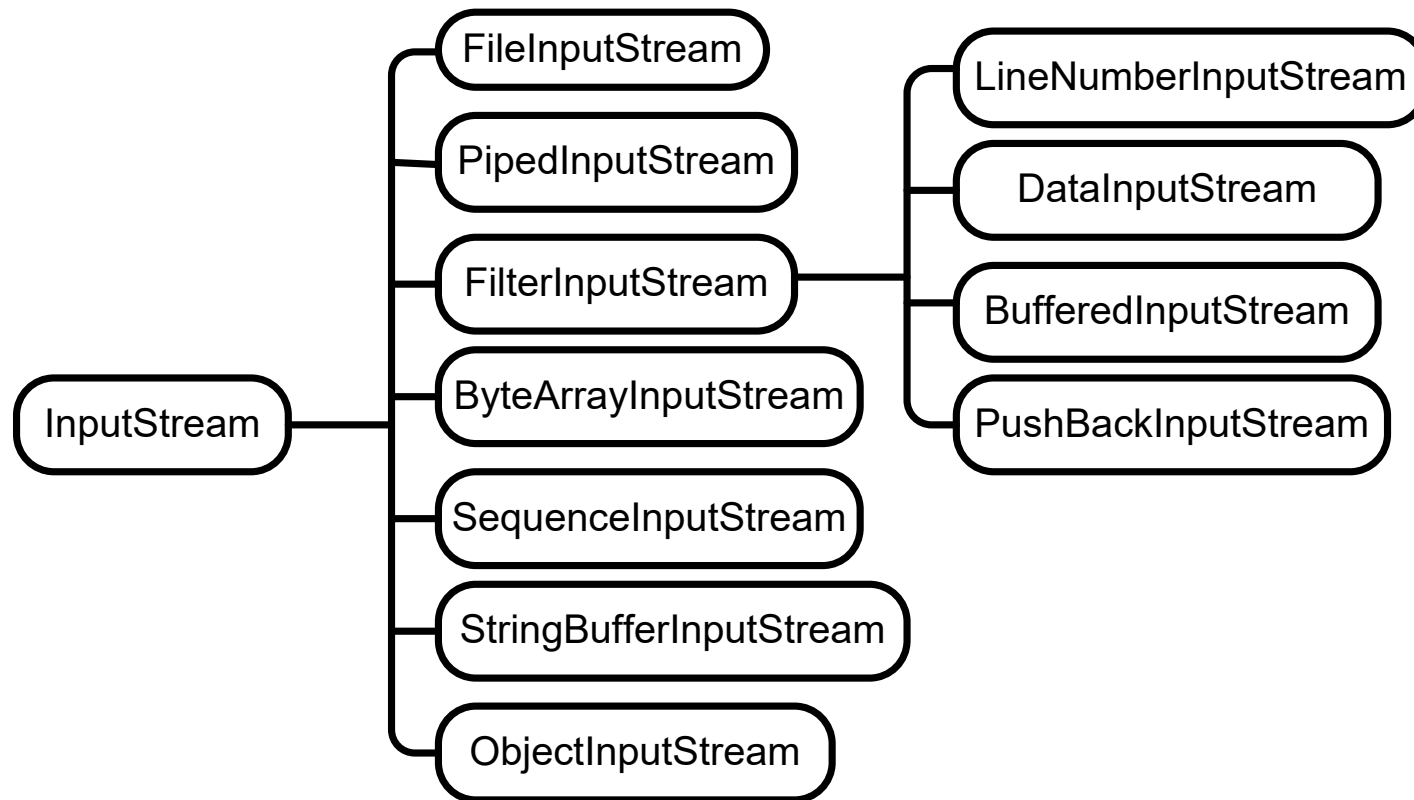
- Ils servent à lire ou écrire des données «bruts» ou binaires

■ Les flux de caractères

- Ils servent à lire ou écrire des données qui représentent des caractères lisibles (caractères Ascii).

Les flux d'octet

■ Flux d'entrée d'octets (1)



Les flux d'octet

■ Flux d'entrée d'octets (2)

● Les principales méthodes de la classe `InputStream`

▶ *`abstract int read() throws IOException`*

↳ Lit le prochain octet. La valeur de l'octet est retournée comme un entier entre 0 et 255 ou -1 si la fin de fichier est atteinte.

▶ *`int read(byte[] b) throws IOException`*

↳ Lit au plus `b.length` octets et les copie dans le tableau `b`. Le nombre d'octets lu est retourné ou -1 si la fin de fichier est atteinte.

▶ *`int read(byte[] b, int offset, int len) throws IOException`*

↳ Idem que précédemment, mais les octets sont stockés à partir de l'indice `offset`. Au plus `len` octets lus

▶ *`void close()`*

↳ Ferme proprement le flux.

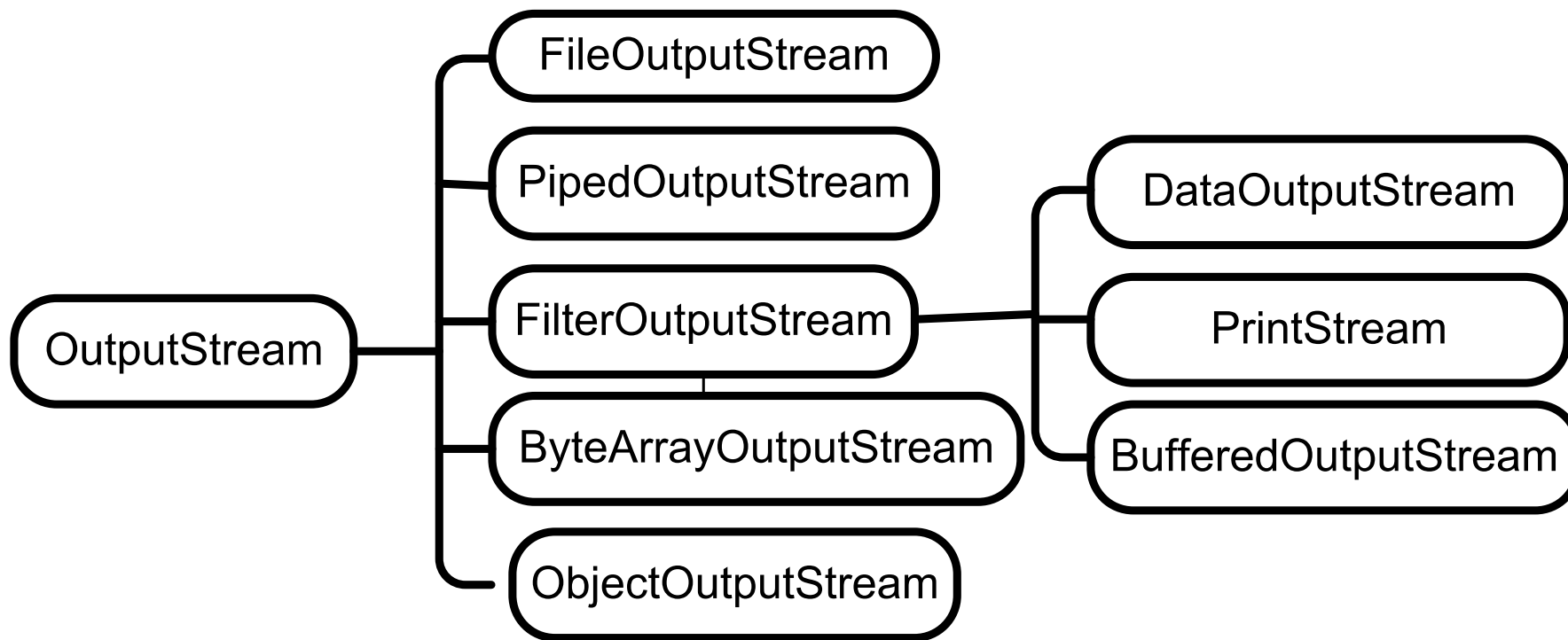
Les flux d'octet

■ Flux d'entrée d'octets (3)

- Les méthodes *read* sont bloquantes au cas ou il n'y a rien à lire
- La méthode *available()* peut être utilisée pour éviter les blocages.

Les flux d'octet

■ Flux de sortie d'Octets (1)



Les flux d'octet

■ Flux de sortie d'Octets (1)

- Les principales méthodes de la classe `OutputStream`

- ▶ `void write(int b)`

- ↳ Ecrit l'octet `b` (seulement les 8 bits de poids faibles sont pris en compte).

- ▶ `void write(byte[] tb)`

- ↳ Ecrit les octets du tableau d'octets `tb`.

- ▶ `int write(byte[] tb, int offset, int len) throws IOException`

- ↳ Ecrit les octets du tableau à partir de l'indice `offset`. Au plus `len` octets lus

- ▶ `void close()`

- ↳ Ferme proprement le flux.

Les flux d'octet

■ Atelier 1

Dans cet atelier on souhaite manipuler les flux d'entrée-sortie. Les classes utilisées sont *FileInputStream* pour le flux d'entrée et *FileOutputStream* pour le flux de sortie. Les méthodes utilisées sont les méthodes de base (*read* ou *write*). Les constructeurs des classes mentionnées sont:

```
FileInputStream(String nomFichier) throws FileNotFoundException;;
```

```
FileOutputStream(String nomFichier) throws FileNotFoundException;
```

Ecrire un programme qui permet, dans un premier temps, de saisir une suite de caractère dans un fichier, puis de lire cette suite de caractère à partir du fichier et l'afficher à l'écran. Il faut noter que la classe *Scanner* ne prévoit pas la lecture direct de caractère. Pour cela, il faut lire d'abord une chaîne de caractère, puis extraire le premier caractère de cette chaîne en utilisant la méthode *charAt(int index)* de la classe *String*.

Les flux d'octet

■ Atelier 2

Ecrire un programme qui permet, dans un premier temps, de saisir des chaînes de caractères dans un fichier, puis de lire ces chaînes à partir du fichier et les afficher à l'écran.

Quelques méthodes utiles de conversion de chaîne de caractères en byte et vis versa

```
String(byte[] bytes)
String(byte[] bytes, int offset, int length)
string.getBytes()
```

Les flux d'Octets

■ Les flux décorateurs ou filtres (1)

- Elles permettent d'ajouter des fonctionnalités supplémentaires à un flux de base:
 - ▶ Utilisation d'un buffer pour optimiser les opérations de lecture et d'écriture
 - ▶ Codage ou décodage des données manipulées
 - ▶ Compression ou décompression des données
 - ▶ Sérialisation des données
 - ▶ Ajout de nouvelles méthodes plus faciles à utiliser
 - ▶ Redéfinir des méthodes de base
- Exemple
 - ▶ `BufferedInputStream`, `BufferedOutputStream`
 - ▶ `PrintStream`
 - ▶ `Scanner`

Les flux d'octet

■ Les flux décorateurs ou filtres (2)

- Un flux décorateur est associé à un flux de base à partir de son constructeur
- Le flux de base devient un paramètre du flux décorateur
- Un flux décorateur peut être un flux de base pour un autre flux décorateur

Les flux d'octet

■ Exemple de flux décorateur pour un flux d'entrée

● La classe Scanner

- ▶ Elle est destinée à la lecture de données à partir d'un flux d'entrée *InputStream*
- ▶ Elle offre nombreuses méthodes: `nextInt()`, `nextLine()`, `nextDouble`, `nextLong`
- ▶ Exemple

```
Try {  
    FileInputStream fis = new FileInputStream("prog.java");  
    Scanner entree = new Scanner(fis);  
    String ligne = entre.nextLine();  
    System.out.println(ligne);  
} catch (FileNotFoundException
```


Les flux d'octet

■ Exemple de flux décorateur pour un flux de sortie

● La classe PrintStream

- ▶ Ce flux décorateur associé à un flux de sortie offre de nouvelles méthodes plus faciles d'utilisation: *print()* et *println()*
- ▶ Exemple

```
Try {  
    PrintStream sortie = new PrintStream(new FileOutputStream("sortie.txt"));  
    sortie.println("Ce texte sera écrit dans le fichier");  
} catch (FileNotFoundException e){}
```

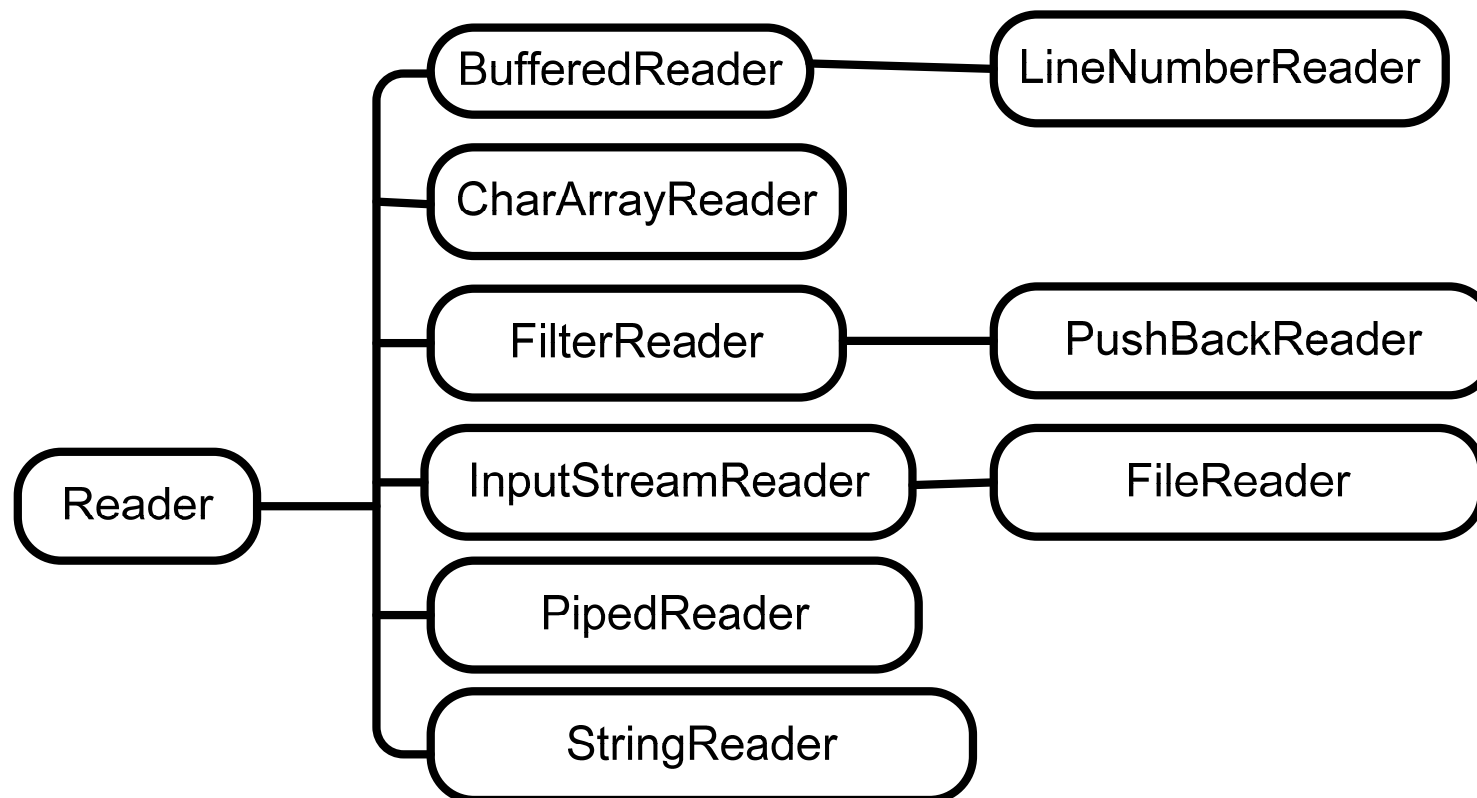
Les flux d'octet

■ Atelier 3

- 1) Ecrire et tester un programme qui permet de lire des lignes de texte au clavier et les écrit dans un fichier. La saisie s'arrête lorsqu'on rentre une ligne vide. La méthode `equals(String s)` de la classe `String` permet de comparer une chaîne de caractère à une autre.
- 2) Ecrire et tester un programme qui permet de lire des lignes de texte à partir d'un fichier et les écrit à l'écran.
- 3) Ecrire et tester un programme qui lit des lignes de fichier à partir d'un fichier et les écrit dans un autre

Les flux de caractères

■ Flux d'entrée de caractère



Les flux de caractères

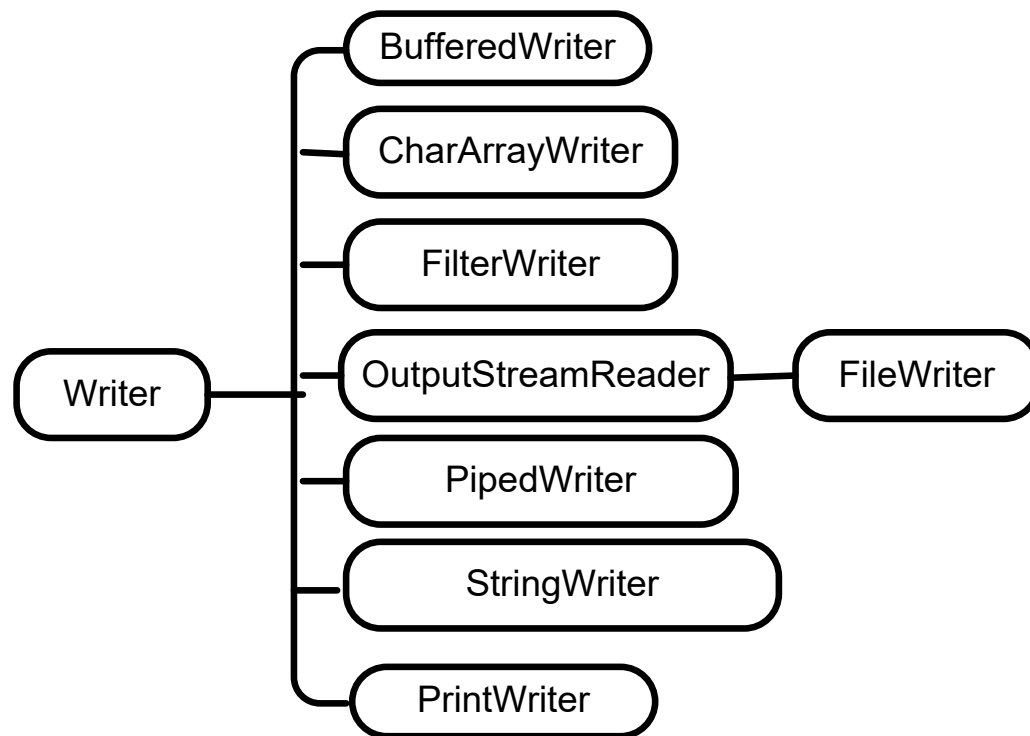
■ Flux d'entrée de caractère

- Les principales méthodes

- ▶ `abstract int read()` throws `IOException`
- ▶ `int read(char[] b)` throws `IOException`
- ▶ `int read(char[] b, int off, int len)` throws `IOException`

Les flux de caractère

■ Flux de sortie de caractère



Les flux de caractères

■ Flux d'entrée de caractère

● Les principales méthodes

- ▶ `abstract void write () throws IOException`
- ▶ `void write(char[] b) throws IOException`
- ▶ `void write(char[] b, int off, int len) throws IOException`
- ▶ `void write(String s)`
- ▶ `void write(String str, int off, int len)`
- ▶ `void flush()`

Les flux d'octet

■ Atelier 4

Ecrire et tester un programme qui permet copier le contenu d'un fichier dans un autre de deux manières:

- 1) Caractère par caractère
- 2) Bloc par bloc

Les flux de caractère

■ Flux décorateur d'entrée de caractère

● La classe `BufferedReader`

- ▶ En plus du tampon d'entrée, elle offre des méthodes plus facile d'utilisation telle que la méthode: *String readLine() throws IOException*
- ▶ Exemple

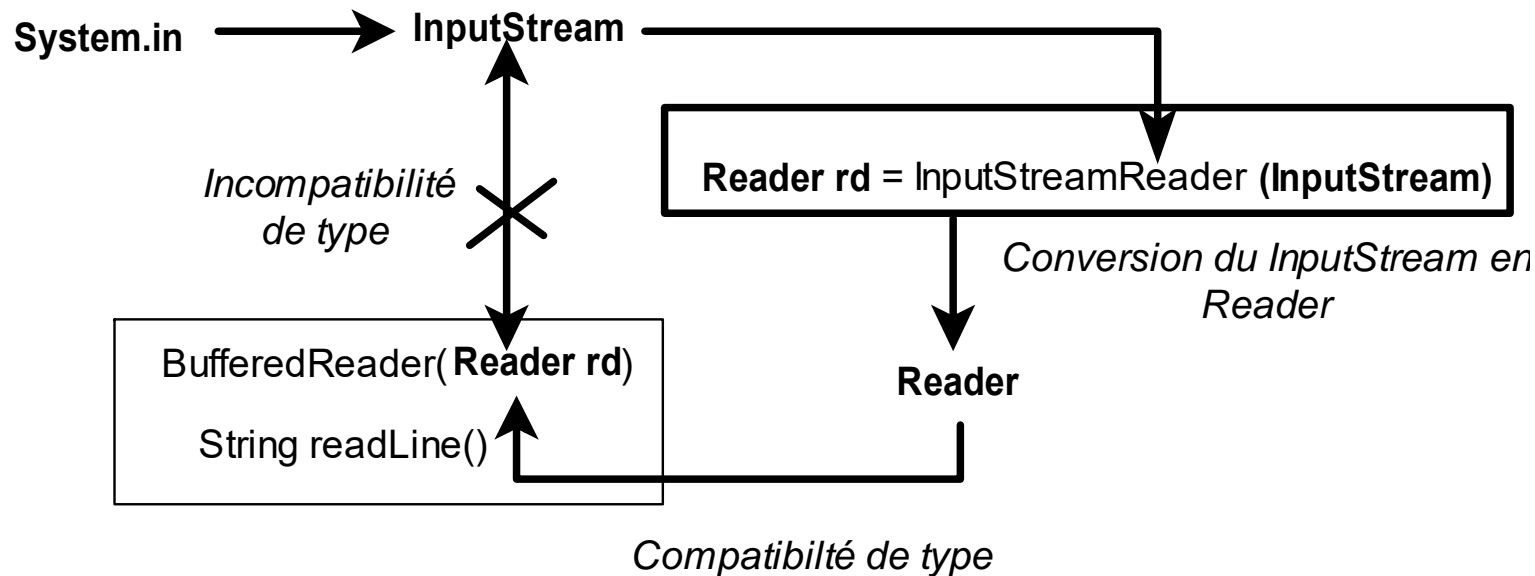
```
try {  
    Reader rd = new FileReader("prog.java");  
    BufferedReader br = new BufferedReader(rd);  
    String ligne = br.readLine();  
}  
catch(FileNotFoundException e) {}  
catch(IOException e) {};
```


Les flux de caractère

■ Flux décorateur d'entrée de caractère

● La classe InputStreamReader

- ▶ C'est un décorateur qui sert de pont entre un flot de caractère et un flot d'octets
- ▶ Il lit des flots d'octets et les convertit en flots de caractère.



Flux de Caractère

■ Flux décorateur d'entrée de caractère

- La classe InputStreamReader

```
Reader reader = new InputStreamReader(System.in);  
BufferedReader clavier = new BufferedReader(reader);  
System.out.print("Entrez une ligne de texte:");  
String line = clavier.readLine();  
System.out.println("La ligne saisi est :" + line);
```

Les flux de caractère

■ Flux décorateur de sortie de caractère

● La classe `PrintWriter`

- ▶ Comme la classe `PrintStream` en terme de méthodes Elle dispose également des méthodes *print*
- ▶ Quelques constructeurs
 - ↳ `PrintWriter(File file)`
 - ↳ `PrintWriter(OutputStream out)`
 - ↳ `PrintWriter(String fileName)`

▶ Exemple

```
try{  
    PrintWriter sortie=new PrintWriter(new FileOutputStream("sortie.txt"));  
    sortie.println("texte à ecrire");  
} catch(FileNotFoundException e){};
```

Flux de Caractère

■ Atelier 5

- 1) Ecrire un programme qui permet de lire le contenu d'un fichier existant ligne par ligne en utilisant les classes *FileReader* et *BufferedReader*
- 2) Ecrire un programme qui lit des lignes de caractères au clavier et les écrit dans un fichier. La classe *BufferedReader* sera utilisé à la place de la classe *Scanner*.

Gestion des fichiers et répertoires

■ La classe `java.io.File`

- Cette classe représente un chemin textuelle vers un fichier ou un répertoire

- Les constructeurs de la classe `File`

- ▶ `public File(String repertoire) throws NullPointerException;`
- ▶ `public File(String repertoire, String fichier)`
- ▶ `public File(File repertoire, String fichier);`

- Exemple

```
File repertoire = new File("c:/java/TP");  
File fichier1 = new File(repertoire, "prog.java");  
File fichier2 = new File("c:/java/TP", "prog.java");
```

Gestion des fichiers et répertoires

■ Les méthodes de la classe *File*

● Méthodes de parcours

Méthodes	Utilisation
<code>String[] list()</code>	Fichiers et répertoires du répertoire.
<code>File[] listFiles()</code>	Fichiers et répertoires du répertoire.
<code>static File [] listRoots()</code>	Liste les répertoires racines (C:\,D:\, E:\,...)

● Méthode de création de fichier ou répertoire

Méthodes	Utilisation
<code>boolean createNewFile()</code>	créé un nouveau fichier vide s'il n'existe pas
<code>boolean mkdir()</code>	créé un nouveau répertoire

Manipulation des fichiers: classe File

■ Les méthodes de la classe *File*

● Les méthodes de test

Méthodes	Utilisation
<code>boolean canRead()</code>	teste le fichier est lisible
<code>boolean canWrite()</code>	Teste si le fichier est modifiable
<code>boolean exists()</code>	teste si la référence existe
<code>boolean isDirectory()</code>	teste si l'objet est un répertoire
<code>boolean isFile()</code>	teste si l'objet est un fichier
<code>boolean isHidden()</code>	teste si l'objet est caché
<code>boolean isAbsolute()</code>	teste si la référence est absolue

Manipulation des fichiers: classe File

■ Les méthodes de la classe *File*

● Les méthodes de d'obtention de propriété

Méthodes	Utilisation
<code>String getAbsolutePath()</code>	renvoie la référence absolue de l'objet
<code>String getName()</code>	Dernier élément de la référence
<code>String getParent()</code>	Référence du répertoire parent
<code>Long lastModified()</code>	Date de dernière modification
<code>Long length()</code>	Longueur de l'objet

Gestion des fichiers et répertoire

■ Atelier 6

- 1) Ecrire un programme qui affiche la liste des répertoires racine d'un système de fichiers
- 2) Ecrire un programme qui saisit le nom d'un répertoire au clavier et affiche la liste des éléments de ce répertoire
- 3) Ecrire un programme qui saisit le nom d'un répertoire au clavier et affiche la liste le type de chacun des éléments (fichier ou répertoire)

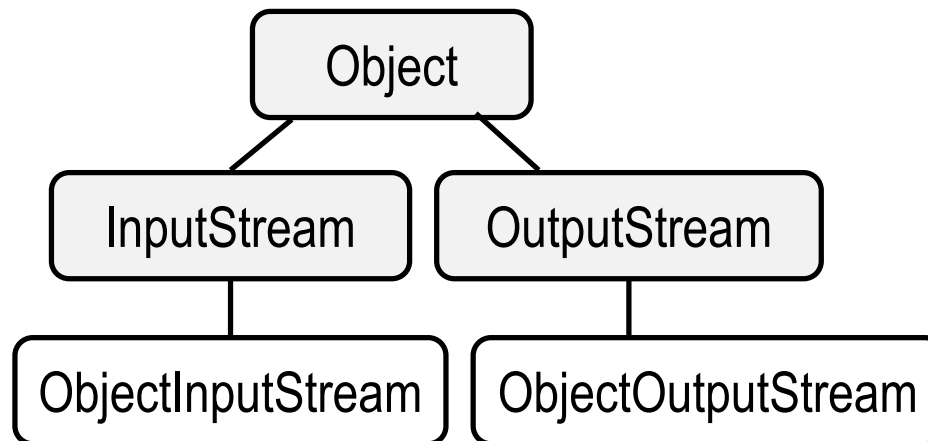
La sérialisation des objets

■ Définition

- La sérialisation est le processus de transformation d'un objet en flux d'octets.
- Il consiste à écrire les attributs de l'objet sur un flux de sortie binaire
- La désérialisation d'un objet est le processus inverse
- La classe de l'objet à sérialiser doit implémenter l'interface *java.io.Serializable*.
- Tout attribut es sérialisé si:
 - ▶ Il est de type primitif ou est une référence d'un objet dont la classe est sérialisée
 - ▶ Il n'est pas déclaré « static »
 - ▶ Il n'est pas déclaré « transient »

La sérialisation des objets

■ Les classes de sérialisation



La sérialisation des objets

■ Constructeurs et autres méthodes

- La classe *ObjectOutputStream*

`ObjectOutputStream(OutputStream out)` throws `IOException`

`private void writeObject(Object o)` throws `IOException`

- La classe *ObjectInputStream*

`ObjectInputStream(InputStream in)` throws `IOException`

`Object readObject()` throws `IOException`, `ClassNotFoundException`

La sérialisation des objets

■ Exemples

- Sérialisation d'un objet

```
FileOutputStream fos = new FileOutputStream("etudiant.dat");  
ObjectOutputStream oos = new ObjectOutputStream(fos) ;  
Etudiant etudiant = new Etudiant("MARI","Konate",38);;  
oos.writeObject(etudiant) ;
```

- Désérialisation d'un objet

```
FileInputStream fis = new FileInputStream("etudiant.dat");  
ObjectInputStream ois = new ObjectInputStream(fis) ;  
Etudiant e2 = (Etudiant)ois.readObject() ;
```

La sérialisation des objets

■ Atelier 6

Cet atelier vise à mettre en pratique le mécanisme de sérialisation des objets

1) Ecrire une classe Etudiant sérialisable, comprenant les attributs suivants:

- nom
- Prénom
- Age
- Statut (boursier ou pas)

Cette classe, en plus du constructeur, cette classe sera dotée d'une méthode «void afficher()» pour l'affichage à l'écran d'un objet Etudiant.

2) Ecrire une classe dans laquelle un objet Etudiant est sérialisé dans un fichier, puis restauré. Faire des tests en affichant l'objet Etudiant avant la sérialisation et après la restauration