Université Paris Est Créteil
IUT de Créteil-Vitry
DUT R&T 2ème année

Module M2207
Consolidation des bases de la programmation
Langage Java

Y. AMIRAT

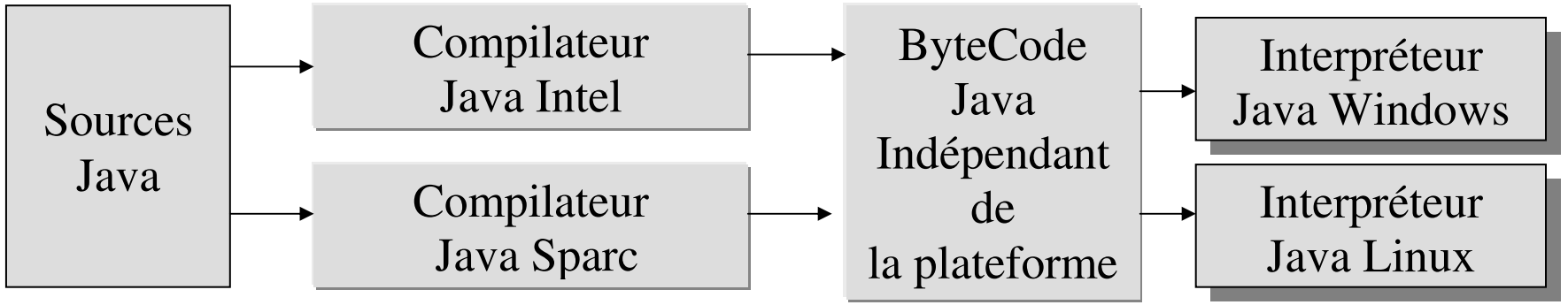
1. Introduction

1.1. Java, un langage de programmation révolutionnaire

Multi-plate-forme (**Write Once, Run Anywhere):** partout où fonctionne l'environnement d'exécution Java (**Machine Virtuelle Java -MVJ**), il est possible d'utiliser des programmes Java.

Neutre du point de vue architectural : Un programme Java pourra s'exécuter sur n'importe quelle plate-forme : Macintoch, PC, LINUX.

Interprété et d'une grande efficacité : La compilation d'un programme Java génère un code intermédiaire appelé ***bytecode***. Ce dernier peut ensuite être interprété sur n'importe quel système doté d'un interpréteur de programme (ou MVJ) approprié. La MVJ est capable d'exécuter un fichier compilé, sans recompilation.



Objet orienté : Opportunité d'organiser les développements (logiciels) de manière à obtenir le maximum de modularité et de réutilisation du code.

Simple et puissant : Java permet d'exprimer chaque idée d'une manière claire et précise, orientée objet, en masquant les éléments de bas niveau, spécifiques aux processeurs et aux systèmes d'exploitation.

Fiable : Java ne permet pas d'accès incontrôlé sur le système de fichiers ou le contenu de la mémoire. Même la communication avec d'autres ordinateurs est contrôlée.

Multithreading : Java permet nativement de répartir plusieurs opérations en des processus parallèles légers (threads).

Robuste:

- **Grâce au mécanisme d'exception**: Les erreurs difficiles à retracer, qui se produisent dans des situations d'exécution particulières sont faciles à prendre en compte dans une application. Exemples d'exceptions : division par zéro, fichier introuvable, serveur hors service.

- Grâce au **mécanisme automatisé de gestion mémoire** appelé récupération de mémoire (**Garbage Collector** ou ramasse-miettes), le programmeur n'a plus à se soucier des problèmes d'allocation et de libération mémoire.

Extensible : Il est possible de lier du code Java avec d'autres langages, par exemple C ou C++, et ainsi d'utiliser des programmes existants : Appel à des **méthodes natives**.

Exécution dynamique

- La machine virtuelle Java exécute le bytecode et permet **le chargement de code additionnel à la volée** : Applications Internet
  - Java permet de **transporter des objets** (au sens de la programmation objet) entre différents programmes : solutions pour l'optimisation et la répartition de charge.

Richesse des bibliothèques

Nombreuses bibliothèques : Structures de données, Entrées/sorties, Interfaces Graphiques, Applications réseaux et Web (Servlets, applets, Web services), Bases de données, Sécurité, etc.

1.2. Un premier programme Java

```
//Notre première application Java : ceci est un commentaire
public class HelloWorld {
      public static void main(String args []){
                  System.out.println("Hello World") ;
      }
}
```

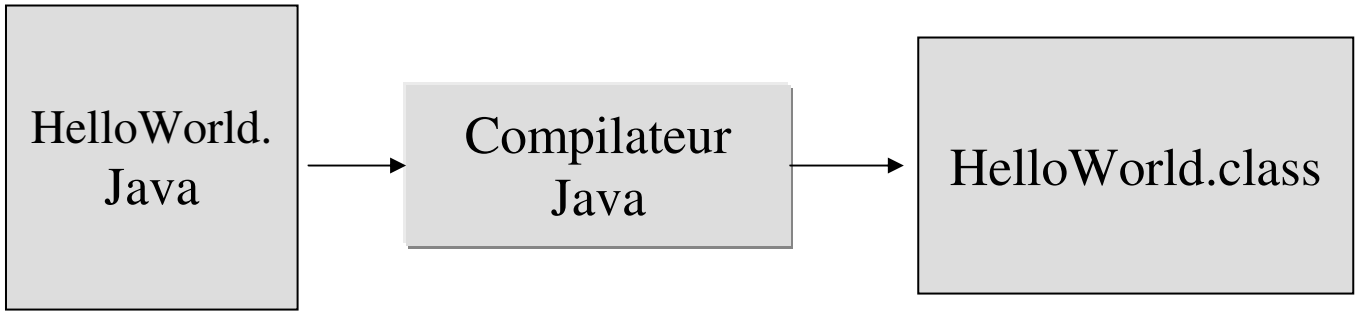
-main() : méthode (fonction ou traitement) principale de la classe. Elle constitue le point d'entrée du programme. Elle est appelée par la commande
>java HelloWorld

-void : caractérise les fonctions qui ne retournent rien (void en Anglais)

-le paramètre String args []: correspond aux paramètres éventuellement transmis en ligne de commande (de la même manière que pour les scripts Linux vus en M1105)

-**System.out.println("HelloWorld"):** Affiche sur la sortie standard (out) de la machine (System) la chaine de caractères "HelloWorld".

Compilation : javac HelloWorld.java



• La méthode main est la méthode qui est automatiquement appelée par l'interpréteur java à l'exécution du fichier HelloWorld.class:

Exécution : java HelloWorld.class

2. CONCEPTS DE LA PROGRAMMATION OBJET

Objet :

Un **objet** est une entité informatique qui regroupe des **données** et des **traitements** permettant de les manipuler.

Exemple

Objet de Type <<Personne>>	
Données (champs) de l'objet	Traitements
Nom **Société**	**PresenteToi** **InitialiseToi**

Intérêts

1. Meilleur organisation et plus grande clarté des programmes.

2. Les programmes orientés-objets sont plus faciles à comprendre que les autres et donc plus faciles à réutiliser, à maintenir et à faire évoluer.

3. Protection des données par **encapsulation**:

4. Un objet P n'est plus un simple agrégat de données mais un objet qui rassemble dans une même entité (on dit qui **encapsule) des variables** (**données**) et des **comportements (traitements)**. Toute exécution d'un traitement faisant intervenir P devra correspondre à l'un des **traitements** encapsulés.

5. Le code gagne ainsi en sécurité car l'objet contrôle chacun des **comportements** qui lui sont propres.

6. Ecriture et évolutivité plus facile de gros programmes grâce aux concepts d'héritage et de surcharge des traitements.

Classe:

Une classe est un moule à partir duquel les objets sont formés. Une classe est une implémentation d'un type d'objet.

Un objet est une **instance** de classe. Il s'agit d'une réplique unique du modèle de la classe, dotée du point de vue des données de son propre ensemble de données appelées **variables d'instances** : Un objet peut être assimilé à une variable structure, la classe étant le modèle de cette structure.

Exemple précédent:

La classe pourrait avoir pour nom **Personne**; Une instance (ou objet) de cette classe pourrait être **Dupont, Martin,** etc.

Définition d'une classe:

Avant de pouvoir créer des objets, il faut définir le moule dans lequel ils seront fondus.

La définition d'une classe en Java a la structure suivante:

Importations de classes ou de packages
class NomDeLaClasse
{ ***définitions des variables d'instance et de classe***
définitions des méthodes d'instance et de classe
}

Remarques:

-Les définitions des éléments de la classe (variables et méthodes peuvent intervenir dans n'importe quel ordre)

-A un moment donné, une **variable d'instance** existera en autant d'exemplaires qu'il y a d'objets instanciés.

-Une **variable de classe** est une variable qui existe en un seul exemplaire pour la classe, quelque soit le nombre d'objets instanciés. Les **variables de classes** permettent d'implémenter une notion objet de **variables globales**.

-Une méthode d'instance **mi** décrit un traitement (comportement) qui ne peut être déclenché qu'en envoyant un message à un objet **obj** de la classe:

obj.mi(paramètres d'appel)

Exemple: Dupont.PresenteToi();

-Une méthode de classe **mc** décrit un traitement qui est en général déclenché par un appel qualifié avec le nom de la classe **CCC**:

CCC.mc(paramètres d'appel)

Exemple: y=Math.sin(x);

Exemples d'utilisation des méthodes de classes:

-méthodes que l'on compte utiliser souvent et où la création d'objet n'est pas nécessaire.

-méthodes agissant sur des variables de classes

-méthodes agissant sur des types primitifs (int, float, ...): méthodes de la classe Math (Math.sin, cos, sqrt, ...=

-une variable de type nomDuType se définit, avec ou sans valeur d'initialisation, de la manière suivante:

spécificateurs nomDuType nomDeVariable;
spécificateurs nomDuType nomDeVariable=valeurInitiale;

-L'expression **spécificateurs** désigne une suite, éventuellement vide, de mots-clés Java qui indiquent l'accès et la catégorie de la variable:

private: variable privée, accessible uniquement dans les méthodes de la classe ,

public: variable publique, sans restriction d'accès,

static: variable de classe: une variable définie avec l'attribut **static** est toujours une variable de classe,

final: variable dont la valeur sera invariable: une telle variable doit obligatoirement être définie avec une valeur. Un attribut **static final** est une constante de classe.

Une variable d'instance ou de classe est privée ou bien publique. En l'absence de spécificateurs, l'accès est de type **package.**

Les notions de package et d'héritage introduisent d'autres spécificateurs d'accès (voir les parties héritage et package).

Exemple

```
class Cercle {
        static int nb_cercles=0;
        static final double PI=3.14159;
        private double x, y, r;

        //suivent ensuite les méthodes de la classe
}
```

nb_cercles: variable de classe, elle est commune à toutes les instances de la classe

PI: constante de classe

x, y: position du centre; r: son rayon: (x, y, r: variables d'instance accessibles uniquement dans les méthodes de la classe Cercle)

-Une méthode de nom **nomMéthode** se définit de la manière suivante:

```
spécificateurs typeRenvoyé nomMéthode (définitions DesParamètres Formels) {
        codeDeLaMéthode
}
```

Les spécificateurs utilisés pour les variables peuvent s'appliquer pour les méthodes et ont la même signification.

Exemple: Le format de définition d'une classe en Java est le suivant:

```
class Personne {
        private String nom, societe;

        public void presenteToi() {
                System.out.println(" Je m'appelle "+nom);
                System.out.println(" Je travaille à "+societe);
        }
```

```
    public void initialiseToi(String leNom, String uneEntreprise){
        nom=leNom;
        societe=uneEntreprise;
    }
}//classe Personne, première version
```

Remarques:

-nom et **societe** sont des **variables d'instance**

-presenteToi () et **initialiseToi** () sont des **méthodes d'instance**

-En Java, il est possible de donner le même nom à plusieurs méthodes différentes (On parle dans ce cas de **surcharge)**, à condition que les **signatures** soient différentes. Une **signature** comprend: le **nom de la classe**, le **nom de la méthode** et le **type des paramètres formels**.

Créer un objet:

La syntaxe est la même que pour déclarer une variable en C.

```
Nom_Classe nom_objet;          //ici on spécifie le nom de l'objet et sa    classe
nom_objet=new Nom_Classe();    //ici on instancie l'objet
```

Exemple

```
Personne p1;                   //p1 désigne un objet personne
p1=new Personne();             // objet instancié
```

Autre possibilité : Personne p1= new Personne();

Remarques:

-L'objet p1 possède donc sa propre variable **nom** et sa propre variable **societe**.

-La première ligne (**Personne p1**) définit la variable p1 mais ne désigne pas encore un objet, la machine virtuelle Java lui attribue la valeur **null** (indétermination)

-L'opérateur **new** est appelé pour instancier un objet de la **classe Personne** (**p1=new Peronne()**). Cet opérateur renvoie ensuite une **référence sur l'objet crée.** La **référence** est **affectée à** la variable **p1,** qui désigne désormais l'objet Personne qui vient d'être instancié.

-En Java, tous les objets sont instanciés par allocation dynamique et la machine virtuelle Java gère l'espace mémoire sans que le programmeur ait à s'en occuper. Une **référence Java** contient l'adresse de l'objet qu'elle désigne, mais le programmeur n'a pas à se soucier des mécanismes d'allocation et de libération mémoire.

Utilisation d'un objet:

Comment déclencher un comportement (ou une méthode d'instance) sur un objet?

Objet . méthode (paramètres);

Exemple:

```
p1.nom="MARTIN";
p1.societe="Renault";
p1 . presenteToi ();     //envoie le message presenteToi à l'objet p1
```

Exemple

```
//fichier Test.java
class Personne {
      //class Personne précédente
}
public class Test {
      public static void main(String args[]) {
            Personne p1;                        //p1 désigne un objet personne
            p1=new Personne();                  // objet instancié
            p1.initialiseToi("DURAND","RENAULT");
            //autre possiblité si nom et société étaient public: p1.nom="DURAND";
            //p1.societe="RENAULT";
      }
}
```

Remarque: Quand une méthode d'instance est lancée à partir d'un objet, celui-ci est constitue **l'objet courant** (ou **instance courante**). La référence sur cet objet est la référence spéciale **this**.

Exemple:

```
public class Cercle {
      double x, y, r;      // x, y: position du centre;  r: son rayon

      double init (double x, double y, double r) {
            this.x=x;this.y=y;this.r=r;
       }
      double circonference() {
            return 2*Math.PI*this.r;          //ceci est plus précis que: return 2*3.14159*r;
      }
      double surface() {
            return Math.PI *this.r*this.r;
      }
      public static void main(String args[]) {
            Cercle c1=new Cercle();
            c1.init (0,0,5);
            System.out.println("Circonférence de c1="+c1.circonference());
            System.out.println("Surface de c1="+c1.surface());
      }
}
```

Constructeur : une méthode particulière

En Java, aucune variable d'instance ne peut être indéterminée: si elle n'est pas initialisée lors de sa définition, la MVJ lui attribue une valeur par défaut en fonction de son type

Exemple de la classe Personne étudiée précédemment : par exemple **null** pour une variable de type String.

```
Personne p1;                    //p1 désigne une instance de la classe Personne
p1=new Personne();              // instanciation de p1
```

Comme la classe Personne n'est pas dotée de constructeurs, à l'instanciation de p1, les valeurs d'instance nom et societe sont initialisées à null (nom et société sont deux références String).

Cette instanciation par défaut peut ne pas convenir aux spécifications de la classe et le programmeur peut définir lui-même l'instanciation qui lui convient en définissant un ou plusieurs **constructeurs**.

Un **constructeur** est une **méthode spéciale, qui spécifie les opérations à effectuer à l'instanciation.**

-Un constructeur n'a pas de type de retour.

-L'identificateur d'un constructeur est le même que celui de la classe.

-La déclaration des paramètres et du corps d'un constructeur suit la forme classique de la définition des méthodes.

-Si la classe dispose de plusieurs constructeurs, leurs **signatures** doivent être différentes. On parle dans ce cas de **surcharge** (ou de **redéfinition**) de méthodes.

Principe: au moment de la création d'un objet, on utilise l'opérateur new. Celui-ci doit faire appel à un constructeur. Si dans la définition de la classe, aucun constructeur n'a été défini, alors, c'est le constructeur par défaut qui est appelé. Sinon le constructeur appelé est déterminé par la forme de l'appel (signature)

Exemple:

```
//ici la classe Personne est dotée de deux constructeurs
class Personne{
      private string nom;
      private string societe;

      public Personne (String leNom) {                              //constructeur 1
            nom=leNom;société=new String("?");
      }

      public Personne (String leNom, String uneEntreprise) {        //constructeur 2
            nom=leNom;société=uneEntreprise;
      }

      public void presenteToi()  {
            System.out.println(" Je m'appelle "+nom);
            System.out.println(" Je travaille à "+societe);
      }
}
```

```
//Classe de test de la classe Personne
public class Test{
      public static void main(String args[]){
      Personne p1,p2;
       p1=new Personne("DIAZ");      // instanciation de p1 et appel au constructeur1
      p2=new Personne("DIAZ","ELF"); //instanciation de p2 et appel au constructeur2
      p1.presente.Toi();
      p2.presente.Toi();
      }
}
```

Remarques:

-Chaque classe Java possède au moins un constructeur (défini dans la classe ou hérité)

-Il y a une utilisation spéciale du mot clé **this** dans le cas des constructeurs. Il permet à un constructeur d'invoquer un autre constructeur de classe.

Le principe de redéfinition peut être étendu à des méthodes autres que des constructeurs :

```
class Point {
   int x, y;

   Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   double distance(int x, int y) {
      int dx = this.x - x;
      int dy = this.y - y;
      return Math.sqrt(dx*dx + dy*dy);
   }
   double distance(Point p) {
      return distance(p.x, p.y);
   }
}
```

```
public class PointDist {
   public static void main(String args[]) {
      Point p1 = new Point(0, 0);
      Point p2 = new Point(30, 40);

      System.out.println("p1 = " + p1.x + ", " + p1.y);
      System.out.println("p2 = " + p2.x + ", " + p2.y);
      System.out.println("p1.distance(p2) = " + p1.distance(p2));
      System.out.println("p1.distance(60, 80) = " + p1.distance(60, 80));
   }
}
```

Héritage: Le mécanisme d'héritage rattache les unes aux autres, de façon hiérarchique, des classes qui ont un certain degré de parenté. De cette manière, les descendants d'une classe héritent de toutes les variables et méthodes de leurs ancêtres tout en ayant la possibilité de disposer de leurs propres variables et méthodes.

Syntaxe :

```
class NomDeClasse1 extends NomDeClasse2 {
                    //corps de la classe NomDeClasse1
}
```

Principe: Le mot **extends** permet d'indiquer que NomDeClasse1 est une **sous classe** directe de NomDeClasse2. On dit aussi que NomDeClasse2 est la **super classe** (directe) de NomDeClasse1.

Ceci permet aux objets de la classe NomDeClasse1 d'hériter des variables et des méthodes de la classe NomDeClasse2. **En Java, pour une classe il ne peut y avoir qu'une super classe directe.**

Exemple:

```
class Forme {
      double x,y;

      Forme(int a;int b) {
            x=a;       y=b;
      }
      void affichePosition() {
            System.out.println(x) ;
            System.out.println(y);
      }
}
class Cercle extends Forme {
      double r;

      double circonférence() {
            return 2*Math.PI*r;
      }
```

```
        double surface() {
                return Math.PI *Math.pow(r,2) ;
        }
}
public class Test {
        public static void main(String args[]) {
                 Cercle c= new Cercle();
                c.x=10;  c.y=20;  c.r=1.0;
                double s= c.surface();
                c.affichePosition();
        }
}
```

Le désignateur super

Le désignateur **super** peut être utilisé dans deux cas :

- Pour faire directement appel aux méthodes de la super classe.

Exemple : **super.translate(x, y) ;**

Cette instruction signifie qu'on appelle la méthode **translate()** de la super classe sur cette instance.

- Pour faire directement appel aux constructeurs de la super classe.

Exemple :

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

```
    double distance(int x, int y) {
        int dx = this.x - x;
        int dy = this.y - y;
        return Math.sqrt(dx*dx + dy*dy);
    }
    double distance(Point p) {
        return distance(p.x, p.y);
    }
}

class Point3D extends Point {
    int z;
    Point3D(int x, int y, int z) {
        super(x, y);
        this.z = z;
    }
```

```
    double distance(int x, int y, int z) {
        int dx = this.x - x;
        int dy = this.y - y;
        int dz = this.z - z;
        return Math.sqrt(dx*dx + dy*dy + dz*dz);
    }

    double distance(Point3D other) {
        return distance(other.x, other.y, other.z);
    }

    double distance(int x, int y) {
        double dx = (this.x / z) - x;
        double dy = (this.y / z) - y;
        return Math.sqrt(dx*dx + dy*dy);
    }
}
```

```
public class Point3DDist {
   public static void main(String args[]) {

      Point3D p1 = new Point3D(30, 40, 10);
      Point3D p2 = new Point3D(0, 0, 0);
      Point p = new Point(4, 6);
      System.out.println("p1 = " + p1.x + ", " + p1.y + ", " + p1.z);
      System.out.println("p2 = " + p2.x + ", " + p2.y + ", " + p2.z);
      System.out.println("p = " + p.x + ", " + p.y);
      System.out.println("p1.distance(p2) = " + p1.distance(p2));
      System.out.println("p1.distance(4, 6) = " + p1.distance(4, 6));
      System.out.println("p1.distance(p) = " + p1.distance(p));
   }
}
```

Affichage : p1=30, 40, 10
p2=0, 0, 0
p=4, 6
p1.distance(p2)=50,9902
p1.distance(4, 6)=2,23607
p1.distance(p)=2,23607

Les exceptions

Les exceptions sont utilisées pour gérer des erreurs ou des situations inhabituelles. Les exceptions sont des objets qui sont crées lors de situations d'erreurs et qui sont traités dans des sections réservées à cet effet.

Avantages :

-Plus grande robustesse des applications
-Gestion fine des erreurs : facilité de mise au point des programmes
-Séparation du traitement des erreurs de celui des situations normales
-Les objets exceptions peuvent comporter de nombreuses informations quand cela est nécessaire.
-Le caractère objet des exceptions autorise la création de hiérarchies d'exceptions. une exception de lecture d'une donnée au clavier peut être traitée comme telle ou comme une erreur générale d'entrées-sorties.

Exemple d'exceptions : division par 0, accès à un élément d'un tableau en dehors des bornes, rupture d'une connexion réseau, etc.

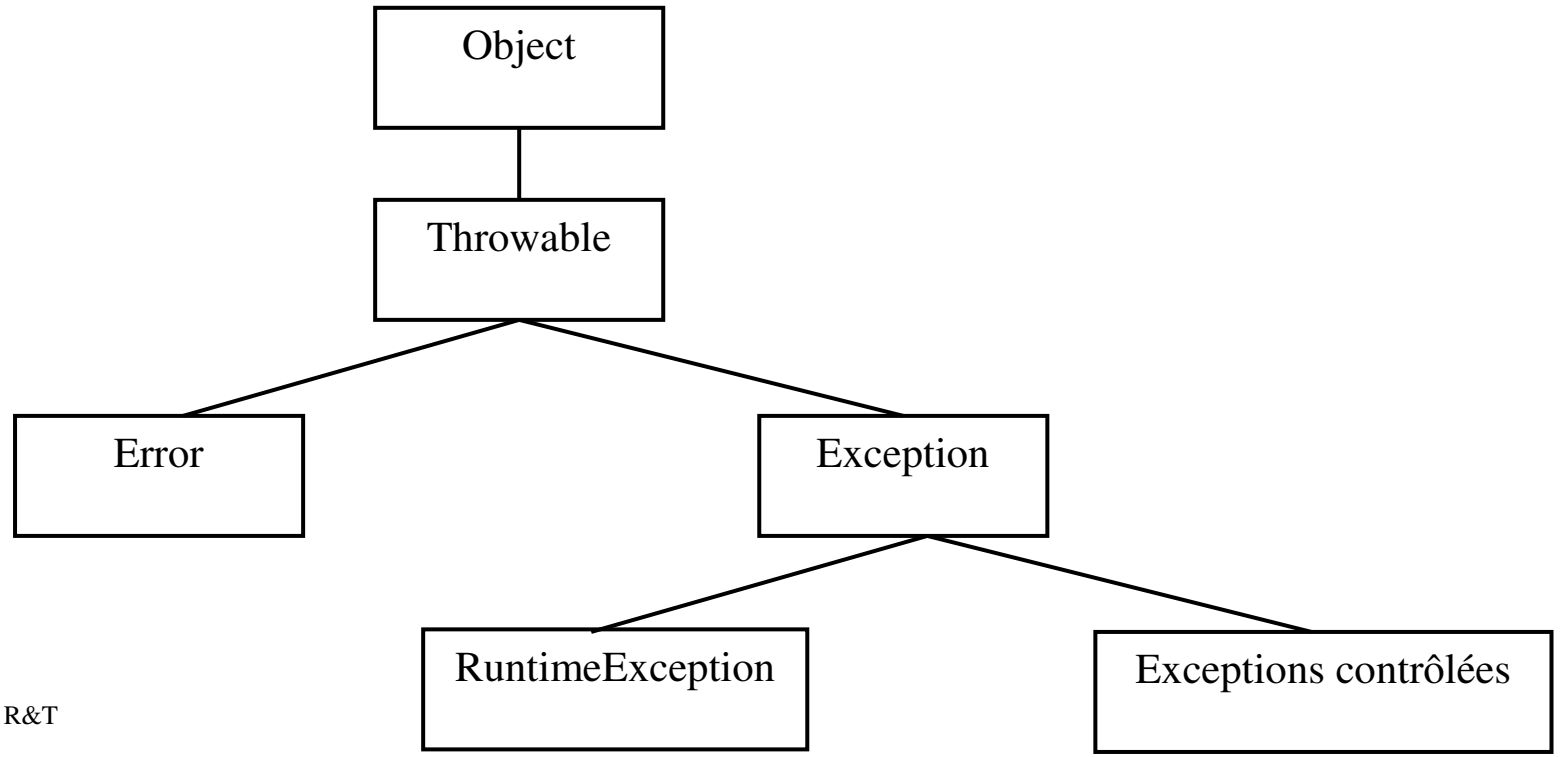
Gestion des exceptions en java : Essentiellement, il s'agit d'essayer (**try**) d'exécuter un bloc de code et, si une erreur se produit, le système lève ou lance (**throws**) une exception qu'on peut essayer d'intercepter (**catch**) selon le type d'exception dont il s'agit ou qui sera finalement (**finally**) traitée par défaut. La levée d'une exception provoque instantanément la sortie du bloc concernée et doit être capturée dans un bloc **catch**.

```
try {
      //appel de méthodes (ou instructions) pouvant lever des exceptions
}
catch (Type-d'exception1 e){
      //gestion de l'exception de Type-d'exception1
}
catch (Type-d'exception2 e){
      //gestion de l'exception de Type-d'exception2
}
…
finally{       //portion de code toujours exécutée
               //généralement utilisée pour permettre un nettoyage après une clause try
}
```

Hiérarchie des classes d'exceptions : On distingue deux types d'exception :

-Les **exceptions non contrôlées** : Exceptions qui n'exigent pas d'être capturées.
-Les **exceptions contrôlées** : Exceptions qui nécessitent d'être capturées par le programme.

- Une exception contrôlée est un objet dont la classe dérive de **Throwable** et **Exception**.
- Les exceptions non contrôlées se subdivisent en deux catégories **Error** et **RuntimeExcepti on.**



La hiérarchie **Error** :

La classe **Error** est la super-classe d'une hiérarchie d'erreurs graves et non contrôlées, qu'il est fortement déconseillé de sous-classer et qu'on peut capturer, mais avec prudence.

Exemples : **NoSuchMethodError**: la méthode référencée n'est pas accessible.

StackOverflowError: il y a un débordement de pile.

OutOfMemory: l'allocation demandée est impossible par manqué de mémoire disponible.

IllegalAccessError: une tentative d'accès à un champ ou une méthode est interdite.

La hiérarchie **RuntimeException** :

La classe **RuntimeException** est la racine d'une hiérarchie d'exceptions non-contrôlées. On peut capturer ces exceptions mais il est déconseillé de les sous-classer.

Exemples : **ArithmeticException** : une erreur arithmétique, comme une division par 0.

NullPointerException : un accès à un objet a été tenté par une référence valant null.

NumberFormatException : une chaîne de caractères est incorrecte pour représenter un nombre.

IndexOutOfBoundsException : l'indice d'un tableau est en dehors des bornes autorisées.

Exemple

```
public class Exc2 {
      public static void main(String args[]) {
      try {             int d = 0;
                  int a = 42 / d;
      }
      catch (ArithmeticException e) {
            System.out.println("division by zero");
      }
      }
}
```

Principe de captage

1. Lorsqu'une instruction génère une exception, le système crée un objet de la classe **Exception**
2. L'exception se propage :
    - vers les blocs englobants
    - vers les méthodes appelantes
3. L'exception est capturée

Si la méthode m() peut produire une exception de type E alors il faut soit :

- Appeler m() dans un bloc try

```
try obj.m() ;
catch (E e) recupération ;
```

- Déclarer que la méthode courante peut générer une exception E

```
void maMethode() throws E {     …
                                obj.m() ;
                                …
}
```

L'exception non captée est propagée à la méthode appelante (ici la méthode qui appelle maMethode() qui doit inclure un bloc **try** ou être déclarée avec l'entête **throws**).

Lorsqu'une méthode lance explicitement une instance de **Exception** ou l'une de ses sous-classes, à l'exception de **RuntimeException**, il faut ajouter dans sa définition le mot clé **throws**

```
type nomdela méthode (liste d'arg) throws liste-d'exceptions {…}
```

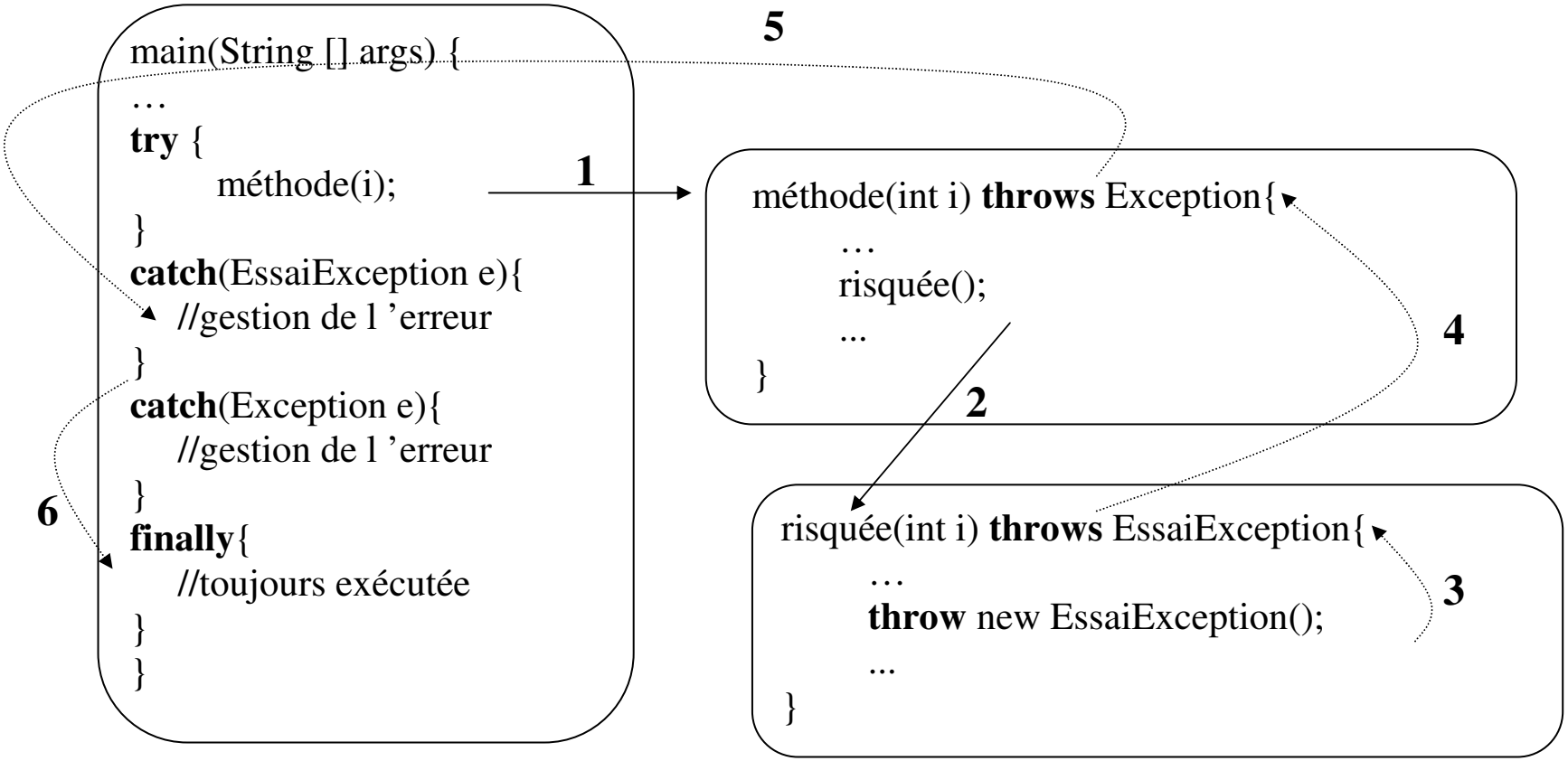
La création d'une nouvelle classe d'exceptions s'obtient par héritage de la classe **Exception**

```
Class NouvelleClasseException extends Exception {
      public NouvelleClasseException() {super() ;}
}
```

Pour lancer explicitement une exception, il faut obtenir une référence d'une instance de **Exception** ou d'une sous classe de **Exception** en créant un objet exception à l'aide de l'opérateur new. Pour lancer l'exception, on utilise l'opérateur **throw** sur l'instance.

```
 throw instance ;
```

Récapitulatif sur le mécanisme de capture des exceptions

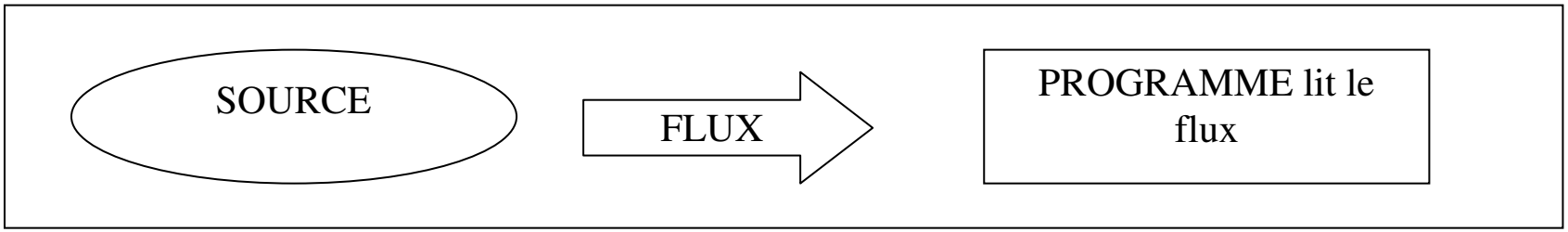


1 et 2: appels de méthodes 3: Levée d 'une exception non capturée
4: Propagation de l 'exception à la méthode appelante. Celle-ci la propage
5: Propagation de l 'exception à la méthode appelante. L'exception est capturée par le premier catch
6: Exécution du bloc finally

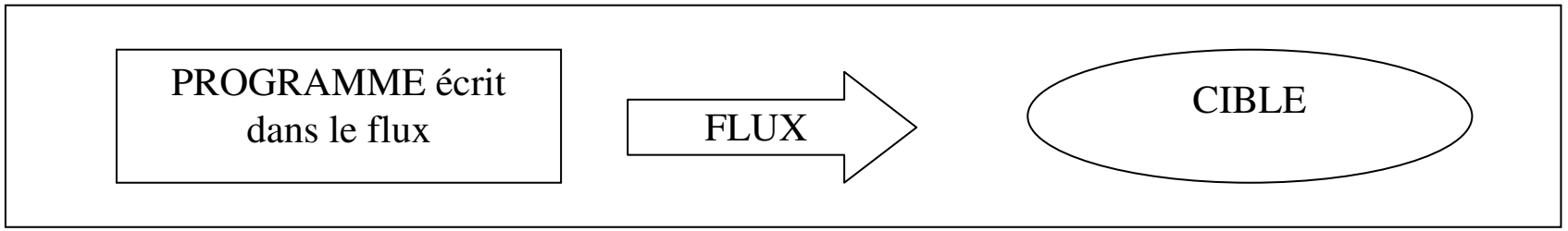
3. LES APPLICATIONS

1. Les flux de données Java

Pour lire une information à partir d'une source de données externe et la récupérer dans un programme, ce programme ouvre un flux (Stream) et lit les informations de manière séquentielle.



Pour transmettre une information à une cible, le programme écrit séquentiellement dans un flux.



Toutes les classes pour la gestion des entrées-sorties se trouvent dans le package **java.io**.

Java utilise deux types de flux :

-flux à accès séquentiel : Les données sont traitées les unes après les autres dans un ordre qui ne peut être changé.

-flux à accès indexés : ce type de flux est utilisé pour traiter des fichiers par des méthodes d'accès direct.

Nomenclature des flux

Dans le package **java.io**, chaque nom de classe est constitué d'un suffixe et d'un préfixe. Un suffixe détermine la nature du flux (**flux de caractères** ou **flux d'octets**) et son sens (**flux d'entrée** ou **flux de sortie**).

	Flux d'entrée (lecture)	Flux de sortie (écriture)
Flux de caractères	**Reader**	**Writer**
Flux d'octets	**InputStream**	**OutputStream**

Suffixes des noms de flux

Les flux se subdivisent en **flux sans traitement de données** et **flux avec traitement de données**.

Dans les **flux sans traitement de données,** le type de la source ou de la destination sert à préfixer le nom du flux.

Source ou destination	Préfixe du nom du flux
Tableau de caractères	**CharArray**
Flux d'octets	**InputStream ou OutputStream**
Chaînes de caractères	**String**
Programme	**Pipe**
Fichier	**File**
Tableau d'octets	**ByteArray**
Objet	**Object**

Préfixes des noms de flux en fonction de la source ou de la destination

Exemple : **InputStreamReader** : flux de caractères, en lecture construit à partir d'un flux d'octets. Cette opération est nécessaire pour lire des chaînes de caractères depuis l'entrée standard **System.in** car elle est de type **InputStream**.

Remarque : Pour la lecture d'un flux d'entrée à partir du clavier, on peut utiliser la classe **Scanner** (du **package java.util**) :

Lecture à partir du fichier standard d'entrée (clavier):

```
Scanner fluxEntree =new Scanner(System.in);
```

Lecture à partir du fichier fluxEntree:

```
Scanner fluxEntree =  new Scanner(new FileInputStream(fichierNom));
```

Dans les flux avec traitement de données, le type du flux d'entrée est généralement le même que celui de sortie, le traitement opère sans changement du type de flux. Le préfixe indique la nature du traitement.

Traitement	Préfixe du nom du flux
Tampon	**Buffered**
Concaténation de flux d'entrée	**Sequence**
Conversion de données	**Data**
Numérotation des lignes de texte	**LineNumber**
Lecture avec retour en arrière	**PushBack**
Impression	**Print**
Sérialisation et désérialisation d'objets	**Object**
Conversion d'octets en caractères (et réciproquement)	**InputStream** ou **OutputStream**

Préfixes des noms de flux en fonction du traitement

Buffered : Ces flux utilisent un **tampon** qui sert de réservoir aux données. En lecture, un périphérique alimente un tampon qu'il remplit et dans lequel on vient lire ensuite. Si le tampon est d'une taille supérieure à celle des données lues à chaque fois, le nombre d'accès au périphérique est réduit. Or les périphériques sont généralement d'un accès plus lent que la mémoire centrale, la réduction du nombre d'accès améliore les performances. En écriture, le fonctionnement est similaire, on écrit dans un tampon qui est ensuite transférée vers le périphérique quand il est plein.

Exemple **:** Lecture d'une chaîne de caractères au clavier :

```
BufferedReader is=new BufferedReader(new InputStreamReader(System.in));
String str=is.readLine();
```

Remarque: System.in, System.out et System.err sont des flux.

Print : Ces flux sont destinés à l'impression, **System.out** est de type **PrintStream**, mais cette classe est dépréciée, il faut utiliser à la place la classe **PrintWriter**.

Toutes les classes de flux décrites ci-avant procurent des méthodes :

-Lecture /écriture d'octets : **read()/write()**
-Lecture/écriture de chaînes de caractères : **readLine()/print()**
-Fermeture du flux : **close()**
-Vidage de buffer : **flush()**

Chaînage de flots

La majorité des classes de flots ont un constructeur ayant un flot en argument :

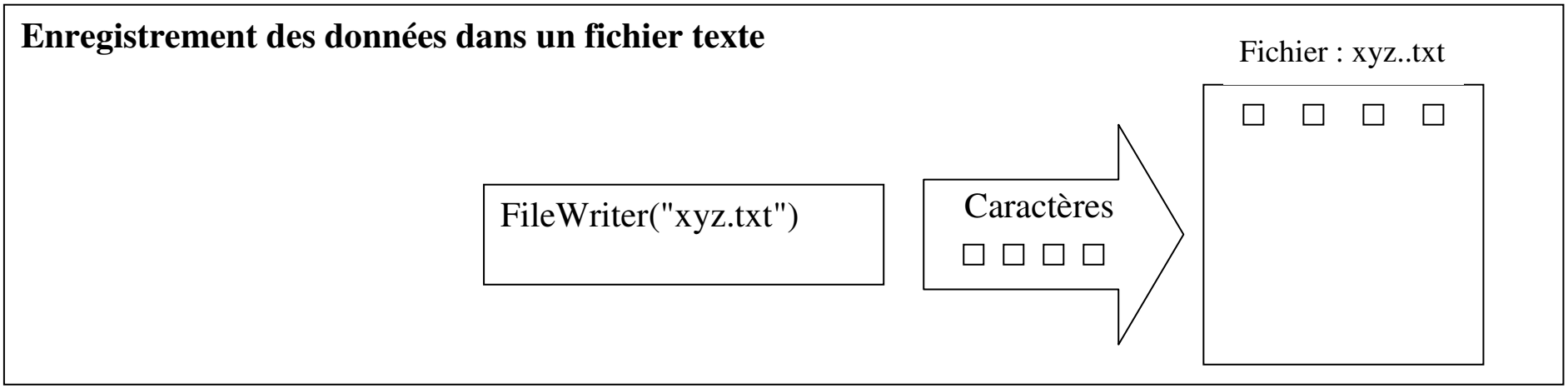
```
pf1=new PremierFlot() ;
pf2=new SecondFlot(pf1) ;
```

Les opérations faites sur un côté sont répercutées automatiquement dans le flot : Lecture, fermeture, etc.

Exemple 1:

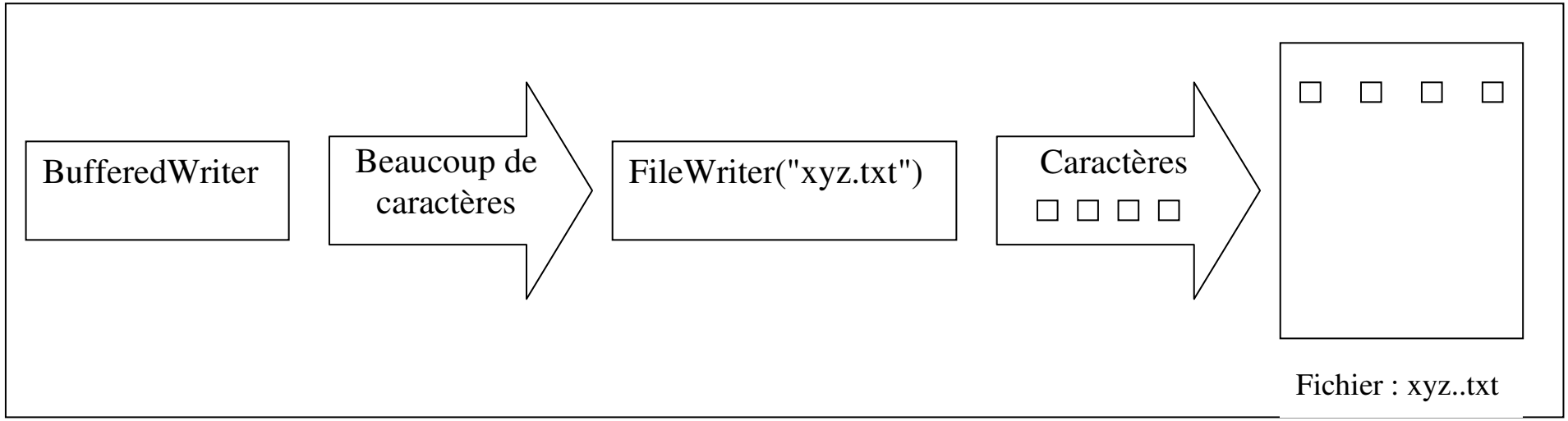
```
public class TestES {
	public static void main(String[] args) {
	Scanner fluxEntree = null;
	PrintWriter fluxSortie = null;
	try {
		fluxEntree = new Scanner(new FileInputStream("original.txt"));
		fluxSortie= new PrintWriter( new FileOutputStream("numerote.txt"));
	}
	catch(FileNotFoundException e) {
		System.out.println("Erreur ouverture fichier.");
		 System.exit(0);
	}
	String ligne = null; int no = 0;
	while (fluxEntree.hasNextLine( )) {
		ligne = fluxEntree.nextLine( );
		no++;
		fluxSortie.println(no + " " + ligne);
	}
	fluxEntree.close( );
	fluxSortie.close( );
 }
}
```

Exemple 2: Enregistrer et lire des données dans un fichier de texte



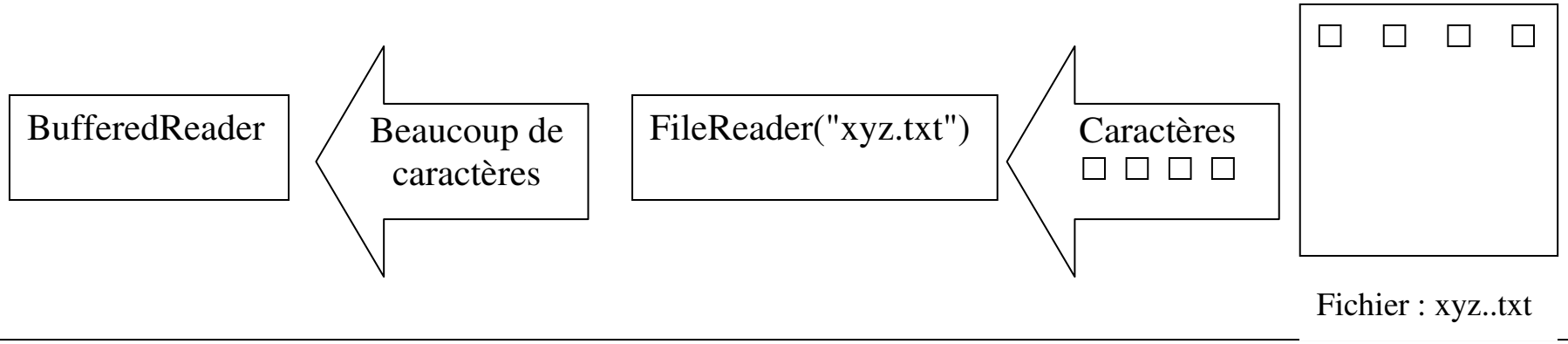

```
try  {     FileWriter out=new FileWriter("xyz.txt");
           out.write("Ceci est un ordre!");
           out.close() ;
}
catch(IOException err) { //origines : disque plein, fichier protégé en écriture, etc.
      System.out.ptintln("Erreur: \n"+err);
}
```

La classe **FileWriter** écrit les données individuellement dans le fichier. Cela peut représenter un gros travail si le volume de données est important, avec pour conséquence, des repositionnements incessants de la tête de lecture du disque. Il est alors judicieux de regrouper les accès en écriture et de procéder à l'écriture effective d'un seul jet. Dans Java, cela s'effectue à l'aide de la classe **BufferedWriter**.



```
try {
        BufferedWriter out = new BufferedWriter(new FileWriter("xyz.txt"));
        out.write("Ceci est un ordre !");
        out.newLine();   //écrit une marque de fin de ligne (saut de ligne) : \r\n
sous Windows et \n sous Unix
        out.write("Et voici l'ordre suivant !");
        out.newLine();
        out.close();
}
catch (IOException err) {
        System.out.println( "Erreur: \n" + err );
}
```

Lire des données à partir d'un fichier texte




```
try {
        BufferedReader in = new BufferedReader(new FileReader("xyz.txt"));
        String ligne;

        while (( ligne = in.readLine()) != null) {
                System.out.println(ligne);
        }
        in.close();
}
catch (FileNotFoundException err) {
        System.out.println( "Erreur : le fichier n'existe pas !\n" + err);
}
catch (IOException err) {
        System.out.println( "Erreur: \n" + err );
}
```

2. La programmation réseau

Le package **java.net** offre des classes pour la programmation réseau. On trouve ainsi :

-des classes permettant d'implémenter des **sockets UDP** (**DatagramSocket**), des **sockets TCP** (**Socket**, **ServerSocket**) et le **multicast** (**MulticastSocket**).

-des classes utilitaires pour:

-l'adressage réseau : adresses IP, services DNS (**InetAddress**)
-l'accès à des ressources (fichiers) par Internet :**URL**, **URLConnexion**)

La classe InetAddress

-Il n'existe pas d'API Java standard permettant de manipuler directement le protocole IP. Seule la classe **InetAddress** est fournie pour encapsuler les manipulations d'adresses et de noms de machines.

- la classe **InetAddress** ne dispose pas de constructeurs. Pour créer un objet **InetAddress**, on dispose de méthodes de classe appropriées :

-getLocalHost : renvoie l'objet de type **InetAddress** qui représente l'hôte local ;

Exemple : afficher l'adresse et le nom de la machine locale

```
InetAddress adresseLocale = InetAddress.getLocalHost();
System.out.println(adresseLocale); // Exemple d'affichage mysotis/194.214.11.207
```

-getByName : renvoie un objet **InetAddress** lorsqu'on lui transmet le nom de l'ordinateur (hostname) concerné.

Exemple : afficher l'adresse et le nom d'un site Web bien connu sur l'Internet

```
InetAddress addresseServeur1 = InetAddress.getByName("www.vuibert.fr");
InetAddress addresseServeur2 = InetAddress.getByName("160.92.127.116");
System.out.println(addresseServeur1);   // Affichage: www.vuibert.fr/160.92.127.116
System.out.println(addresseServeur2);   // Affichage: 160.92.127.116 /160.92.127.116
```

La classe **InetAddress** possède également quelques méthodes d'instance qu'on peut utiliser sur les objets renvoyés par les méthodes décrites précédemment :

-getHostName(): renvoie un String qui représente le nom de l'hôte associé ;

-getHostAddress() : renvoie un String qui représente l'adresse IP de l'hôte;

-getAddress() : renvoie un tableau de quatre octets qui représentent l'adresse IP de l'hôte associé;

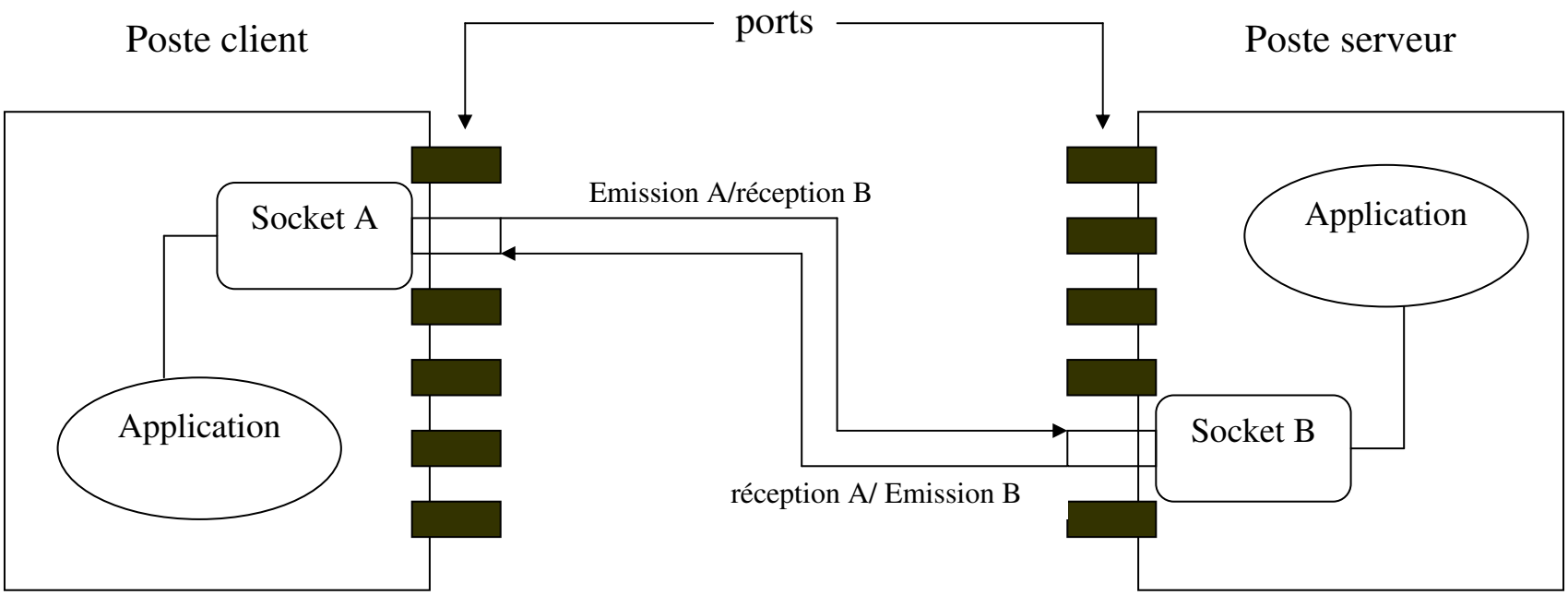
```
InetAddress a = InetAddress.getByName("www.vuibert.fr");
System.out.println(a.getHostAddress());          //affiche : 106.92.127.116
System.out.println(a.getHostName());             //affiche : www.vuibert.fr
```

Remarques : Toutes les méthodes décrites précédemment lancent une exception de type **UnknownHostException** lorsqu'elles ne parviennent pas à retracer la machine correspondant au **hostname**.

Les sockets TCP-IP

Les sockets permettent de créer une liaison dite bidirectionnelle fiable, de type point à point entre deux applications généralement placées sur des postes distants. Définie par une application, une socket est un point terminal de communication du réseau. Pour l'identifier, il faut :

- le nom du serveur sur lequel fonctionne l'application ;
- le numéro de port utilisé pour la transmission des données.



A partir d'un programme Java, on peut utiliser les sockets pour faire communiquer deux applications (ou applets). Les classes **Socket** et **SocketServer** qui sont intégrées au package **java.net** prennent en charge la liaison les échanges de données entre les applications.

Mise en œuvre d'une connexion TCP depuis un programme client :

1. Pour initier la connexion à partir du poste client, il faut créer un objet de la classe **Socket** :

```
Socket sc=new Socket(serveur, port) ;
```

serveur: identification du serveur: adresse IP, nom, instance d'**InetAddress**
port : le numéro de port utilisé lors de la connexion.

Cette instanciation peut lancer une exception de type **UnknownHostException** ou **IOException**.

2. Une fois qu'on a crée l'objet **Socket** (connexion initialisée), on crée les flux d'entrée-sortie auxquels il est associé. Les méthodes d'instance ci-après permettent de réaliser ces opérations:

-getInputStream() renvoie l'objet **InputStream** associé à ce socket
-getOutputStream() renvoie l'objet **OutputStream** associé à ce socket

L'application client se sert ensuite de ces flux pour envoyer ou recevoir des données au moyen de méthodes telles que : **read()**, **print()**, etc. Selon les besoins de l'application, il est possible d'avoir recours à d'autres types de flux (flux de caractères, flux bufférisées, etc.)

3. Lorsque la connexion doit être interrompue, le poste client doit demander au serveur de refermer la connexion avec l'instruction :

```
sc.close() ;
```

Chaque méthode décrite précédemment (**getInputStream(),getOutputStream()** et **close()**) lance une exception de type **IOException** si les sockets sont devenues inutilisables à cause de la rupture d'une connexion réseau.

Exemple : Le code ci-après établit une connexion avec un port de numéro 880 sur un serveur nommé timehost.starwave.com et affiche les données qui sont renvoyées.

```
int c ;
try {           Socket sc = new Socket("timehost.starwave.com",880);
                InputStream in=sc.getInputStream() ;
                while ((c=in.read()) !=-1) System.out.print((char)c) ;
                sc.close();
}
catch (Exception e) {
                System.out.println(e);
}
```

Mise en œuvre d'un programme serveur :

1. Côté serveur, il faut créer un objet de la classe **SocketServer**, correspondant à la socket définie sur le client. Il faut indiquer le numéro de port sur lequel le programme serveur accepte les connexions.

```
SocketServer ss=new SocketServer(port) ;
```

Un **ServerSocket** diffère considérablement d'un **Socket** ordinaire. Lorsqu'on crée un **ServerSocket**, il s'enregistre lui-même dans le système comme recherchant des connexions avec des clients.

2. Il faut appeler la méthode **accept()** sur l'objet ss pour placer le serveur en position d'attente. Ainsi, il va accepter n'importe quelle demande de connexion émise par un poste distant :

```
ss.accept();
```

La méthode d'instance **accept()** arrête l'exécution du programme en attendant qu'un client amorce les communications puis renvoie un **Socket** ordinaire (Socket client).

Les méthodes précédentes (**ServerSocket()** et **accept()** peuvent générer une exception de type **IOException.**

Exemple : Le code ci-après est celui d'un serveur qui attend des connexions sur le port 1000 de l'ordinateur local. Quand un client se connecte à ce serveur, ce dernier lit les données du flux d'entrée provenant du socket et les écrits sur le flux de sortie du socket.

```
try {       ServerSocket ss = new ServerSocket(1000);
            Socket sc = ss.accept();
            BufferedReader in=new BufferedReader(new InputStreamReader (sc.getInputStream()));
            PrintWriter out = new PrintWriter(sc.getOutputStream());
            out.print("Hello! Enter BYE to exit.");
            out.println();
            out.flush();
            boolean done = false;
            while (!done) {
                  String str = in.readLine();
                  if(str == null) done = true;
                  else {      out.print("Echo: " + str );
                              out.println();
                              out.flush();
                              if (str.trim().equals("BYE"))     done=true;
                  }
            }
            sc.close();
}
catch (Exception e) {
      System.out.println(e);
}
```

La classe URL : Accès direct aux ressources Internet

-Les **Locateurs de ressources uniformes**, ou **URL**, permettent d'identifier ou d'adresser des
renseignements sur l'Internet de façon unique, et ce, de manière raisonnablement intelligible.

-Les adresses URL sont utilisées dans tous les navigateurs pour identifier des renseignements sur le Web.

-La classe URL permet de mettre en œuvre la notion d'URL.

Format : protocole://nom_hôte ou adresse_IP:numéro_de_port/nom_du_fichier

Le numéro de port est optionnel

Exemples: http://www.starwave.com/
http://www.starwave.com :8080/index.html.

Etapes nécessaires pour lire un fichier depuis un poste distant sur Internet :

1. Créer un objet URL pour identifier la ressource (le fichier) à charger :

```
URL url=new URL("http://www.serveur.fr/fichier.txt");
```

Une exception de type **MalformedURLException** peut être levée lors de la création de l'URL.

2. Dans la classe URL, une méthode **openStream()** ouvre une connexion avec la ressource située à l'adresse indiquée. Celle-ci retourne un objet de la classe **InputStream**.

```
InputStream in=url.openStream() ;
```

3. Convertir l'objet **InputStream** en **DataInputStream**; il permettra d'accéder aux fonctions de transfert. L'objet **BufferedInputStream** est destiné à améliorer les performances du transfert du fichier en lisant les données par blocs.

```
DataInputStream fichier=new DataInputStream(new BufferedInputStream(in));
```

4. Utiliser les méthodes de l'objet **DataInputStream** pour transférer les données sur le poste destination :

```
String ligne ;
while((ligne=fichier.readLine() !=null) {
   //inclure ici le traitement à effectuer sur chaque ligne transférée
}
```

Une exception de type **IOException** peut être levée par la création des flux **in** et **fichier,** et par la lecture dans le flux **fichier**.

Autres méthodes d'instance de la classe URL : **getProtocol()**, **getPort()**, **getHost()**, **getFile(), toExternalForm().** Le code qui suit en montre un exemple d'utilisation:

```
URL hp = new URL("http://www.starwave.com:8080/people/naughton ");
System.out.println("Protocole: " + hp.getProtocol());   //Affiche Protocole: http
System.out.println("Port: " + hp.getPort());            //Affiche Port: 8080
System.out.println("Host: " + hp.getHost());            //Affiche Host: www.starwave.com
System.out.println("File: " + hp.getFile());            //Affiche File: /people/naughton
System.out.println("Ext:" + hp.toExternalForm());       //Affiche                    Ext:
                                                        //http://www.starwave.com/people/naughton
```

La classe URLConnection

Pour avoir accès aux renseignements (données, bits) contenus dans une **URL**, on crée un objet de la classe **URLConnection** en utilisant la méthode **openconnection**.

Exemple : Ici le programme établit une connexion HTTP sur le port 80 et demande le document qui est le document par défaut (ordinairement index.html). Puis, il dresse la liste des valeurs d'en-tête et en récupère le contenu.

```
int c,i
URL hp = new URL("http","127.0.0.1",80,"/");
URLConnection hpCon = hp.openConnection();
System.out.println("Date: " + hpCon.getDate());                //Date de récupération de la ressource
System.out.println("Type: " + hpCon.getContentType());        //Type de la ressource
System.out.println("Exp: " + hpCon.getExpiration());          //Date d'expiration de la ressource
System.out.println("Last M: " + hpCon.getLastModified());     //Date de dernière modif. de la ressource
System.out.println("Length: " + hpCon.getContentLength());    //Taille de la ressource (octets)
if (i=hpCon.getContentLength() > 0) {                          //Récupération et affichage du contenu
      System.out.println("=== Content ===");
      InputStream input = hpCon.getInputStream();
      while (((c = input.read()) != -1) && (--i > 0))
            System.out.print((char) c);
      input.close();
 }
else
      System.out.println("Aucun contenu disponible");
```

ANNEXES

1. Syntaxe du langage : Les types de base

Types arithmétiques

byte	8 bits	signé	(-128 , 127)
short	16 bits	signé	(-32768 , 32767)
int	32 bits	signé	(-2147483648 , 2147483647)
long	64 bits	signé	(-9223372036854775808, 9223372036854775807)
float	32 bits	signé	(1.4E-45 , 3.4028235E38)
double	64 bits	signé	(4.9E-324 , 1.7976931348623157E308)

Type caractère

char	16 bits	non signé UNICODE2

Type booléen

boolean	1 bit	deux valeurs possibles : true ou false

L'affectation :

-L'opérateur « = » permet d'affecter la valeur de l'expression qui est à droite à la variable qui est à gauche

```
class Essai{
      int calcul(){
            int a = 0;
            int b = 5;
            a = 2*(2*b+2);
            return a+b
      }
}
```

Les opérateurs arithmétiques

-Ces opérateurs peuvent s'appliquer sur les types **entiers** ou **réels :**

- **+, - , * , / , %** (modulo) **, += , -= , *= , /=**

```
int a, b, c;
b=2 ;
c=3 ;

a=b*c ;  // a vaut 6
a +=2 ;  //a vaut 8
b=a/4 ;  //b vaut 2
b=a%2;  //b vaut 0
```

-Les opérateurs **=** et **+=** peuvent être appliqués sur des variables de type « String »

Les opérateurs unaires

-Les opérateurs unaires s'appliquent à un seul opérande de type **entier** ou **réel**

- **-, --, +, ++**

```
int a, b;
a=3;

b = -a ;        // b vaut -3
b = ++a;        //b vaut 4, a vaut 4
b = a--         //b vaut 4, a vaut 3
```

-La pré et la post-incrémentation permettent de réduire le nombre de lignes de byte code

Les opérateurs de comparaison

-Les opérateurs de comparaison s'appliquent sur des **entiers, booléens, réels :**

==, !=, <=, >, >=

- Ils retournent une valeur du type boolean

```
boolean majeur;
int age;
majeur = (age>=18);
```

Les opérateurs logiques

-Les opérateurs logiques s'appliquent au type boolean
  - ! (not) , && (and) , II (or)
  - &, I

-Ils retournent un type boolean

-**&** renvoie « true » si les deux expressions renvoient « true »

-**&&** a le même comportement mais n'évalue pas le seconde expression si la première est « false »

-| renvoie « true » si l'une des deux expressions renvoie « true »

- || a le même comportement mais n'évalue pas le seconde expression si la première est « true »

Les conversions de type

-On dénombre 4 contextes possibles de conversion (cast) :
  - Conversion explicite
  - Affectation
  - Appel de méthode
  - Promotion arithmétique

-Certaines conversions provoquent une perte de valeur

- float en int, int en short, short en byte

-Le type boolean ne peut pas être converti en entier

```
double  f = 3.14;
int  i,j ;
short  s;

i = (int) f;                    // float -> int (conversion explicite)
float ff = (float)3.14;

i = s;                          // short -> int (affectation)

// appel de la method   int obj.m (int i)
obj/m(s) ;                      //short ->int (appel de la methode)

//division d'un entier et d'un flottant : l'entier i est converti en flottant, puis la division
//flottante est calculée
f = i / (double) j ;            // f vaut 0.3333…
```

Test conditionnel

```
class Test {
      public static void main (String args []){
            int cpt = 0;
            boolean start;
            if  (cpt == 0){
                  start = true;
                  System.out.println (“Début de la partie”) ;
            }
            else if (cpt == 10)
                  System.out.println (“Début de la partie”) ;
            else
                  System.out.println (“Début de la partie”) ;
            }
```

Boucle while

```
class Test {
      public static void main (String args []){
            int i = 0;

            do {
                  System.out. println("i=" + i) ;
                  i++;
            }
            while (i<=5);
      }
}
```

Boucle for

```
class Test {
      public static void main (String args []){

            for (int i = 0;i<=5; i++)
                        System.out. println(“i=“ + i) ;
      }
}
```

Switch

```
class Test {
      public static void main (String args []){

            int i=3;
            switch (i) {
            case 1 :  System.out. println(“i=1“) ;
                        break;
            case 2 :  System.out. println(“i=2“) ;
                        break;
            default:  System.out. println(“i ne vaut ni un ni deux“) ;
                        break;

      }
}
```

2. L'initialisation des objets en Java: Que se passe-t-il à la création d'un objet:

1. Toutes les variables sont initialisées par défaut à 0 (ou null)
2. sinon, il est possible de les initialiser par la valeur mise en place:

Syntaxe:

spécificateurs nomDuType nomDeVariable=valeurInitiale;

3. Il est possible d'initialiser des variables via le constructeur invoqué

4. Il est également possible d'initialiser certaines variables via super() (voir l'héritage)

Règles d'accès dans une méthode:

Si à l'intérieur d'une méthode, on utilise un identificateur x, trois cas peuvent se présenter:

-x a été défini comme une variable de la méthode ;
-x est une variable locale;

-x est un champs de l'objet **this.x**, dans ce cas, x est équivalent à **this.x**

Exemple 1: l'instruction x=10 implique les phases suivantes :

1. Le compilateur va vérifier si x est une variable locale ou un paramètre. Si oui, il cherche à affecter 10 à x (test de correspondance de type).

2. Sinon, le compilateur vérifie si **this.x** est admissible. C'est à dire si x est un champ de donnée de l'objet courant **this**.

Exemple 2: l'instruction test()dans le corps d'une méthode implique:

Le compilateur va l'interpréter comme **this.test()**. Il va donc chercher s'il existe une méthode d'instance test() pour l'objet **this**. C'est à dire si dans la classe de l'objet **this**, il y a la définition d'une méthode d'instance test() (voir l'héritage)

3. Passage d'arguments à un programme Java :

Les arguments sont passés à une application Java (**Arguments de la méthode main**) via **un tableau de String**. La longueur du tableau est connu à partir de la méthode **length** des tableaux.

```
public class Syntaxe {
      public static void main(String args[]) {

            //Boucle d'affichage des paramètres du programme
            for (int i=0 ;i<args.length() ;i++){
                  System.out.println("Paramètre N°" +i+args [i]);
            }

            //si on suppose que le premier paramètre est un int (donnée numérique)
            int param1 = Integer.parseInt(args[0]);
            System.out.println("Paramètre N° 1" +param1);
      }
}
```

Valeur de retour d'un programme Java

La méthode main Java ne retourne rien (void). Pour retourner quelque chose il faut lancer la méthode de classe **exit()** de la classe **System** :

Syntaxe : System.exit(valeur).

Variables globales

En Java, il est impossible de créer une **variable globale** en dehors de toutes les classes. Les seules variables qui peuvent être assimilées à des variables globales sont **les variables statiques publiques** dont l'emploi est peu recommandé.

4. **Eléments syntaxiques et structures données Java (voir détails en annexe)**:

-**Types primitifs de Java:** boolean, char, byte, short, int, long, float, double

-Commentaires: /*commentaire*/
// commentaire jusqu'à la fin de la ligne
/** */ pour la génération de documentation

-Les **chaînes de caractères** sont manipulées sous la forme d'**objets de classe String**.

-La **manipulation de tableaux** se fait **par référence**. La **création** (instanciation) d'un tableau se fait au moyen de l'opérateur **new.**

Exemples de création: byte buffer[]=new byte[1024];
byte deuxDim[][]=new byte[256] [256];

Exemples d'utilisation: buffer[1] //élément indicé 1 du tableau buffer
deuxDim[2][3] //élément indicé 2,3 du tableau deuxDim

-Il existe de nombreuses classes utilitaires permettant de créer et de manipuler des structures de données complexes : Vector, Hashtable, etc.

-En ce qui concerne le passage de paramètres lors de l'appel à une méthode:

-Un objet est passé par référence
-Un tableau est passé par référence
-Un type primitif est passé par valeur

-La **null référence** est utilisé pour le contrôle sécurité (objet ou donnée n'ayant aucune référence)

5. Les Packages

Java offre la possibilité de **regrouper des classes** dans un **paquetage** (ou **package**). On utilise les packages lorsque l'on a un nombre important de classes, dans le cas de grandes applications. On peut ainsi les utiliser pour créer une librairie de classes fréquemment utilisées.
Un package peut contenir des classes, des interfaces ou bien un autre package (sous-package)
Si l'on désire ranger une classe dans un package, il faut insérer la ligne suivante au début du fichier source :

```
package nomdupackage ;
```

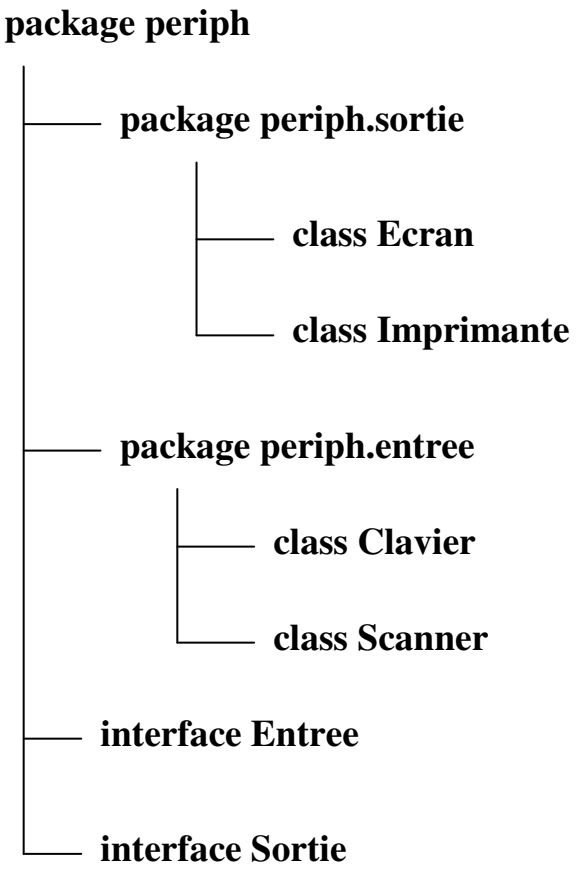
Il est possible de créer une hiérarchie de packages de classes en séparant les niveaux par des points. Cette hiérarchie doit se refléter dans le système de fichiers du système d'exploitation.

```
package nomdupackage1. Nomdupackage2. Nomdupackage3 ;
```

Exemple :

Les classes faisant partie du package periph doivent être placées dans le répertoire periph

Les classes faisant partie du package periph.sortie devront se trouver dans le répertoire periph/sortie (sous Unix) ou periph\sortie (sous Windows)



Remarque : l'utilisation de packages permet de déclarer deux classes de même nom dans des packages différents.
Il existe deux façons pour utiliser les classes d'un package. Soit l'on fait référence au nom complet d'une classe, soit l'on importe la classe.

Exemple :pour créer une instance de la classe clavier de l'exemple précédent, on peut écrire :

```
periph.entree.Clavier leClavier=new periph.entree.Clavier() ;
```

Chaque variable et méthode peut aussi être référencée par son nom complet:

```
nomDuPackage.nomDeLaClasse. nomDeLaMéthode
```

Il est plus simple d'utiliser import :

```
import periph.entree.Clavier ;
…
Clavier leClavier=new Clavier() ;
```

Remarques :

-Pour importer toutes les classes d'un package, on utilise le raccourci suivant :

import nomdupackage.* ;

L'astérisque(*) n'importe pas les sous-packages.

-Le compilateur java importe implicitement le package java.lang dans tous les programmes. C'est l'équivalent de la ligne suivante : **import java.lang.* ;**

Forme générale d'un fichier source java

Un fichier source Java possède l'extension **.java**. A la compilation, un fichier .class est généré pour chacune des classes: **NomDeLaClasse1.class ... NomDeLaClasseP.class.**

```
package nomDuPackage                    //optionnelle unique
import package1.NomDeLaclasse1          //importation de la classe NomDeLaClasse1 du
                                        package //package1
...
import packageN.NomDeLaclasseM
class NomDeLaClasse1{
      ...
}
...
class NomDeLaClasseP{
      ...
}
```

Le langage Java dispose de plusieurs packages qui constituent la librairie standard. La librairie standard Java est un exemple d'arborescence composée de plusieurs niveaux de packages, dont la **racine commune est la package java.**

Exemples: -le **package java.lang**: rassemble des **classes comme**: **Object, System** ou **String**.

-L'instruction **import java.util.Date** permet d'utiliser la **classe publique Date** du **package java.util**.

-L'instruction **import java.util.*** permet d'utiliser **toutes les classes publiques** du **package java.util**.

Remarque : les packages sont parfois rassemblés et compactés dans un fichier d'archive au format ZIP (ou JAR).

La protection d'accès et les modificateurs d'accès

Java offre de nombreux niveaux de protection qui permettent un contrôle rigoureux de la visibilité des variables et des méthodes d'une classe. Etant donnée l'existence des packages de classes et du concept d'héritage, java doit tenir compte de 4 catégories en ce qui concerne la visibilité des éléments des classes (variables et méthodes) :

- Les sous-classes qui appartiennent au même package ;
- Les classes qui appartiennent au même package, mais ne sont pas des sous-classes ;
- Les sous-classes qui appartiennent à des packages différents ;
- Les classes qui appartiennent à des packages différents et qui ne sont pas des sous-classes.

	private	**Pas de modificateur**	**private protected**	**protected**	**public**
Même classe	Oui	Oui	Oui	Oui	Oui
Sous-classes du même package	Non	Oui	Oui	Oui	Oui
Non-sous-classes du même package	Non	Oui	Non	Oui	Oui
Sous-classes de packages différents	Non	Non	Oui	Oui	Oui
Non-sous-classes de packages différents	Non	Non	Non	Non	Oui

La valeur de chaque case précise si l'accès à partir d'un lieu donné (**la ligne**) sera accordé dans le cas d'une variable ou méthode déclarée selon son degré de visibilité (**colonne**)

6. Méthode abstraite

Dans certaines situations, il est nécessaire de définir une classe qui déclare la structure d'un concept abstrait sans fournir une version complète de chaque méthode.

On peut alors déclarer que **certaines méthodes doivent être redéfinies dans des sous classes en employant le modificateur de type abstract**. Le détail (l'implémentation) de ces méthodes est alors de la responsabilité des sous classes.

Toute classe qui contient une ou plusieurs méthodes déclarées **abstract** doit être déclarée **abstract**

Toute sous classe d'une classe **abstract** doit soit implémenter toutes les méthodes **abstract** de la super classe, soit être déclarée elle même **abstract**.

Exemple : A est une classe ayant une méthode abstract (appellemoi) et B une classe qui implémente cette méthode. La classe Test est une classe de test.

```
abstract class A {
abstract void appellemoi ();
   void moiaussi() {
      System.out.println("On est dans la méthode moiaussi de A");
   }
}
class B extends A {
   void appellemoi () {
      System.out.println("On est dans la méthode appellemoi de B ");
   }
}
public class Test {
   public static void main(String args[]) {
      B obj = new B();
      obj.appellemoi ();
      obj.moiaussi();
   }
}
```

7. Les interfaces

Une interface est une forme de classe complètement **abstraite** où :

-Toutes les méthodes de l'interface sont abstraites.
-Tous les champs de données sont static et final (constantes)
-Il n'y a pas de variables d'instance

Contrairement à ce qui se passe dans l'héritage, une classe peut implémenter (définir de manière concrète) n'importe quel nombre d'interfaces. Pour ce faire, il suffit qu'elle possède une version de toutes les méthodes de l'interface. Les méthodes de cette version doivent correspondre aux signatures des méthodes de l'interface.

Syntaxe de déclaration : Il n'est pas nécessaire de rajouter les mots-clés abstract et final dans la déclaration d'une interface.

```
interface nomInterface{
        type_de_retour nom_de_la methode1(liste_de_parametres) ;
        type nom_de_la_variable_final=valeur ;
                        …
}
```

Syntaxe d'utilisation :

```
class nomClasse extends SuperClasse implements Interface1, Interface2 …{
        //coprs de la classe
}
```

Exemple :

```
interface Callback {
   void callback(int param);
}

class Client implements Callback {
   public void callback(int p) {
      System.out.println("callback appelée avec " + p);
   }
}
```

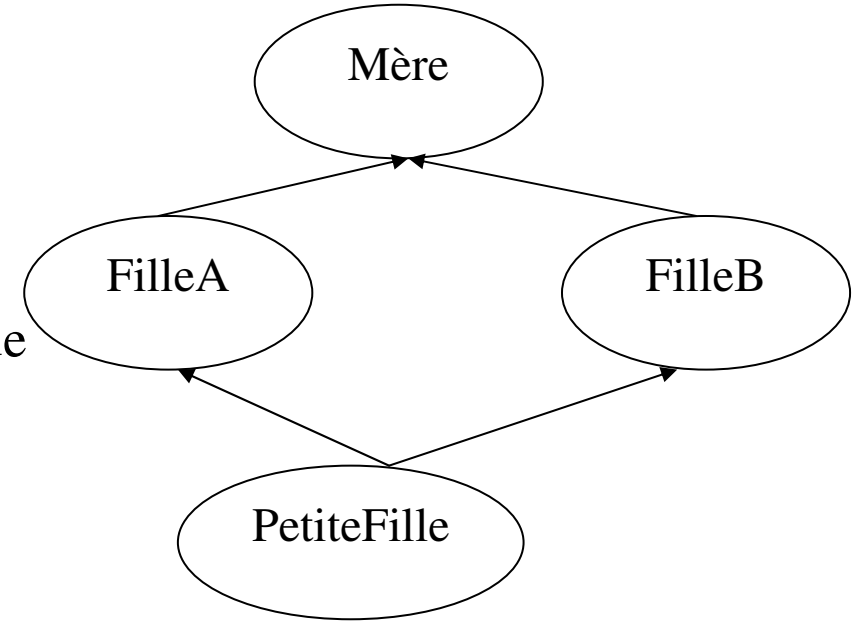
Remarque : Les interfaces appartiennent à une hiérarchie différente de celles des classes. Il est donc possible pour plusieurs classes qui n'ont absolument aucun lien hiérarchique d'implémenter la même interface.

Interface et héritage multiple

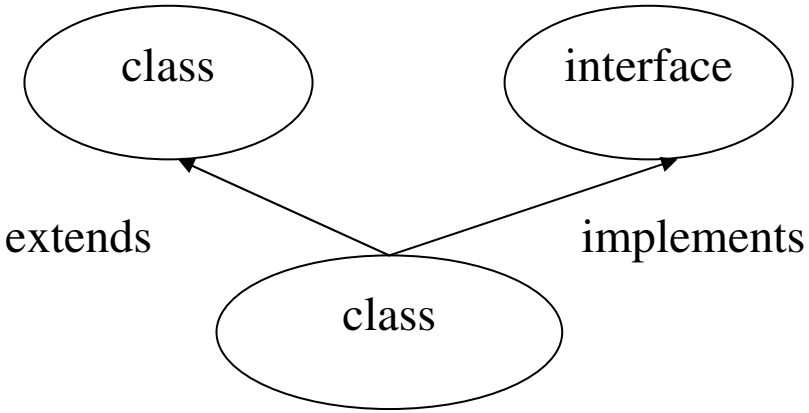
Pourquoi n'y a-t-il pas d'héritage multiple en Java ? Parce que définir un héritage multiple pose de nombreux problèmes, notamment celui de l'héritage en diamant.

-Un attribut qui est défini dans la classe mère
Apparaît-il en double dans la classe PetiteFille ?

-Si une méthode de la classe Mère est redéfinie
dans FilleA et FilleB quelle est celle qui s'applique
dans PetiteFille ?



L'héritage multiple est utile dans un cas classique : lorsque on veut hériter d'une classe qui définit le QUOI et d'une classe qui définit le COMMENT.
Java autorise une classe à hériter d'une superclasse (COMMENT) et à implémenter une interface (QUOI)



Remarques : Il existe une classe **Object** dont héritent toutes les autres classes. Si on ne précise pas de clause d'héritage dans la définition d'une classe, par défaut elle hérite de Object.

Class Heritier{…} est équivalent à **Class Heritier extends Object{…}**

Une interface peut hériter d'une autre interface :

interface Transformable extends Scalable, Rotatable,Reflectabel{…}

Université Paris Est Créteil
IUT de Créteil-Vitry
DUT R&T 2ème année

Travaux Dirigés

Module M2207
Consolidation des bases de la programmation
Langage Java

Y. AMIRAT

TD 1 : Types simples, chaînes de caractères et tableaux, quelques classes utiles

Le but de ce TD est de vous familiariser avec la syntaxe de Java et d'étudier quelques classes utiles.

Exercice 1 : En utilisant l'éditeur Eclipse, écrire un programme Java qui affiche **Bonjour**.

Exercice 2 : Corriger les programmes suivants :

```
public class Scope {
	public static void main(String args[]){
		int i =0;
		for (int i=0 ;i<5 ;i++)  {System.out.print(i+",") ;}
		System.out.print("\n") ;
	}
}
```

```
public class Variables {
	public static void main(String args[]){
		float a=3.0 ;
		double b=4 ;
		float c ;
		c=Math.sqrt(a*a+b*b) ;
		System.out.print("c="+c) ;
	}
}
```

```
public class Promote {
	public static void main(String args[]){
		byte b=42 ;
		char c='a' ;
		short s=1024 ;
		int i=50000 ;
		float f=5.67f,
		double d=.1234 ;
		double resultat=(f*b)+(i/c)-(d*s) ;
		System.out.print((f*b)+ "+"+(i/c)+ "-"+(d*s)) ;
		System.out.println("="+resulat) ;

		byte b2=10 ;
		byte b3=b2*b;
		System.out.println("b3="+b3) ;
	}
}
```

Exercice 3: Soit le fichier source java ci-après :

```
public class Array {
        public static void main(String args[]){
                String jour_semaine[]=new String [7];
                int jour=Integer.parseInt(args[0]) ;
                jour_semaine[0]="dimanche" ;
                jour_semaine[1]= "lundi" ;
                jour_semaine[2]= "mardi" ;
                jour_semaine[3]= "mercredi" ;
                jour_semaine[4]= "jeudi" ;
                jour_semaine[5]= "vendredi" ;
                jour_semaine[6]= "ça me dit" ;
                System.out.println("Moi, le préfère le "+jour_semaine[jour]) ;
        }
}
```

Que se passera-t-il si on lance les commandes :
$java Array 2
$java Array 7
Expliquer.

Exercice 4 : On considère la méthode suivante :

```
1 public static String buildString(String s) {
2       String r="";
3       for(int i=0;i<s.length();i++) {
4               if(keepChar(s.charAt(i),i)) {
5                       r=r+s.charAt(i);
6               }
7       }
8       return r;
9 }
```

On remarque que la méthode utilise la méthode keepChar.

1. On suppose que la méthode keepChar est la suivante :

```
1 public static boolean keepChar(char c,int position) {
2       return true;
3 }
```

Que fait alors la méthode buildString (justifiez votre réponse) ?

2. On suppose que la méthode keepChar est la suivante :

```
1 public static boolean keepChar(char c,int position) {
2       if(position%2==1) {
3               return true;
4       } else {
5               return false;
6       }
7 }
```

Quel est le résultat de la méthode buildString si on lui transmet la chaîne "NBVCX" ?
Que fait la méthode buildString ?

3. Pour chaque résultat souhaité pour la méthode buildString proposez une méthode keep-Char qui permette son obtention :

(a) La chaîne renvoyée par buildString doit contenir un caractère sur trois de la chaîne paramètre, en conservant le caractère de position 1, puis celui de position 4, etc.
L'image de "ABCDEF" est donc "BE".

(b) La chaîne renvoyée par buildString doit contenir les caractères qui ne sont pas des chiffres contenus dans la chaîne paramètre. L'image de "AB2C3D6EF" est donc "ABCDEF".

Vous pourrez utiliser la classe Character qui propose une méthode isDigit[1] qui à un char associe true si et seulement si ce caractère correspond à un chiffre.

[1] Exemple: `boolean test=Character.isDigit('A');`

TD 2 : Notions de classe, objets, Héritage

Exercice 1:

1. Créer un fichier TestPoint.

2. Réaliser une classe ***Point2D*** permettant de manipuler un point d'un plan. On prévoira:
-un constructeur recevant en arguments les coordonnées (float) d'un point,
-une méthode d'instance ***deplace*** effectuant une translation définie par ses deux arguments (***float***),
-une méthode d'instance ***affiche*** se contentant d'afficher les coordonnées cartésiennes du point.
-une méthode d'instance ***String toString()*** permettant d'afficher sous forme textuelle les coordonnées cartésiennes d'un objet ***Point2D***.
Les coordonnées du point seront des attributs privés.

Ecrire par ailleurs, une classe de test ***TestPoint*** déclarant un point, l'affichant, le déplaçant et l'affichant à nouveau. Pour l'affichage, on testera 2 méthodes.

3. Adapter la classe précédente de manière à ce que la méthode de classe ***afficheNbPoints*** affiche le nombre d'objets de type ***Point2D*** créés.

Tester cette nouvelle classe dans la classe de test ***TestPoint***.

4. Réaliser une classe ***Point2Dbis***, analogue à la précédente, mais ne comportant pas de méthode ***affiche***. Pour respecter le principe d'encapsulation des données, prévoir deux méthodes d'instance publiques (nommées ***abscisse*** et ***ordonnée***) fournissant en retour respectivement l'abscisse et l'ordonnée d'un point.
Tester la classe dans la classe de test ***TestPoint***

5. Ajouter à la classe précédente de nouvelles méthodes d'instance:
-***rotation*** qui effectue une rotation dont l'angle est fourni en argument,
-***rho*** et ***theta*** qui fournissent en retour les coordonnées polaires du point.
Tester cette nouvelle classe dans la classe de test ***TestPoint***

Exercice 2 : Soit la classe ***Point2D*** de l'exercice 1 (question 2) .

1. Créer un fichier TestPoint3.

2. Expliquer la différence entre les spécificateurs d'accès **private, private protected** et **protected.**

3. Créer une classe ***Point2Dter***, dérivée de ***Point2D*** comportant une nouvelle méthode nommée **rho**, fournissant la valeur du rayon vecteur (première coordonnée polaire) d'un point.
Quel doit être le spécificateur d'accès pour les variables d'instance x et y de la classe ***Point2D*** pour qu'elles soient accessibles depuis la classe fille ***Point2Dter*** ?

4. Quelles sont les méthodes utilisables pour une instance de type ***Point2Dbter*** ? Ecrire une classe de test TestPoint3

Exercice 3 : Soit la classe ***Point2D*** de l'exercice 1 (question 2).

1. Créer une classe ***Point2DAvecCouleur***, dérivée de ***Point2D***, comprenant :
-un attribut **private** supplémentaire **cl**, de type String, destiné à contenir la « couleur » d'un point.
-les méthodes suivantes :
-**affiche**(redéfinie), qui affiche les coordonnées et la couleur d'un objet de type ***Point2DAvecCouleur***,
-**colorie**(String couleur), qui permet de définir la couleur d'un objet de type ***Point2DAvecCouleur***
-un constructeur permettant de définir la couleur et les coordonnées d'un point

2. Tester cette classe dans la classe de test TestPoint3

Exercice 4 : Soit la classe ***Point2Dbis*** de l'exercice 1 (question 4)

1. Créer un fichier TestPoint2D4 contenant la classe de test TestPoint2D4. Qu'affiche le code de la classe TestPoint2D4 ci-dessous :

```
Point2Dbis p1=new Point2Dbis (1,2);
Point2Dbis p2=p1;
Point2Dbis p3=new Point2Dbis (1,2);
System.out.println(p1==p2);
System.out.println(p1==p3);
```

2. Écrire dans la classe Point2Dbis une méthode ***estIdentiqueA()*** (à vous de trouver la signature exacte de la méthode) qui renvoie true si deux points Point2Dbis ont les mêmes coordonnées cartésiennes.

3. Remarquons qu'il existe déjà une méthode nommé ***equals()*** dans la classe Object dont le rôle est de tester si deux objets sont égaux sémantiquement.
Transformer la méthode ***estIdentiqueA ()*** en méthode ***equals()***.

4. Qu'affiche le code suivant :

```
Object p=new Point2Dbis (1,2);
Object p2=new Point2Dbis (1,2);
System.out.println(p.equals(p2));
```

Expliquer ce résultat. Comment obtenir un résultat plus logique ?

TD 3 : Interfaces-Exceptions

Exercice 1 : Expliquer le comportement des programmes.

1.

```
class Essai1Exception extends Exception{
        Essai1Exception(String s){
                super(s) ;
        }
}
class Essai2Exception extends Essai1Exception{
        Essai2Exception(String s){
                super(s) ;
        }
}
public class Exn{
        static void throwEssais(int i) throws Exception {
                switch (i){
                        case 1:  System.out.println("Lancement de Essai1Exception") ;
                                 throw new Essai1Exception ("Essai1Exception de throwEssais") ;
                        case 2:  System.out.println("Lancement de Essai2Exception") ;
                                 throw new Essai2Exception ("Essai2Exception de throwEssais") ;
                        default: System.out.println("Lancement de Exception") ;
                                 throw new Exception("Exception de throwEssais") ;
                        }
        }
        public static void main(String[] args){
                for (int i = 1 ; i <=3 ; i++){
                        try {
                                 throwEssais(i) ;
                                 System.out.println("Retour d'exception") ;
                        }
                        catch (Essai2Exception e){
                                 System.out.println("Catch Essai2: " + e.getMessage()) ;
                        }
                        catch (Essai1Exception e){
                                 System.out.println("Catch Essai1: " + e.getMessage()) ;
                        }
                        catch (Exception e){
                                 System.out.println("Catch Exception : " + e.getMessage()) ;
                        }
                        finally {
                                 System.out.println("Finally de main.") ;
                        }
                }
        }
}
```

2.

```
class Essai1Exception extends Exception{
        Essai1Exception(String s){
                super(s) ;
        }
}
class Essai2Exception extends Exception{
        Essai2Exception(String s){
                super(s) ;
        }
}
public class Exnbis{
        static void throwEssais(int i) throws Exception {
                switch (i){
                        case 1: System.out.println("Lancement de Essai1Exception") ;
                                throw new Essai1Exception ("Essai1Exception de throwEssais") ;
                        case 2: System.out.println("Lancement de Essai2Exception") ;
                                throw new Essai2Exception ("Essai2Exception de throwEssais") ;
                        default: System.out.println("Lancement de Exception") ;
                                throw new Exception("Exception de throwEssais") ;
                }
        }
        public static void main(String[] args){
                for (int i = 1 ; i <=3 ; i++){
                        try {
                                throwEssais(i) ;
                                System.out.println("Retour d'exception") ;
                        }
                        catch (Essai1Exception e){
                                System.out.println("Catch Essai1: " + e.getMessage()) ;
                        }
                        catch (Essai2Exception e){
                                System.out.println("Catch Essai2: " + e.getMessage()) ;
                        }
                        catch (Exception e){
                                System.out.println("Catch Exception : " + e.getMessage()) ;
                        }
                        finally {
                                System.out.println("Finally de main.") ;
                        }
                }
        }
}
```

TD 4 : Entrées-Sorties

Exercice 1 :

1. Analyser la classe suivante :

```
import java.net.*;
import java.io.*;

public class URL2Fichier {
/* Méthode principale */
public static void main(String[] args) {
   final int TAILLE_TAMPON = 4096;
   URL uneUrl ;
   InputStream fluxE=null ;
   FileOutputStream fluxS=null ;
   BufferedOutputStream donneesEcrites = null;
   BufferedInputStream donneesLues = null;
   int nbLus=0 ;
   byte [] tampon ;                    //tampon de lecture et d'écriture

   if (args.length != 2) {
         System.err.println(" Le programme requiert 2 arguments ");
         System.err.println("l'URL du fichier à copier");
         System.err.println("le nom du fichier destination");
        System.exit(1) ;
      }

      try {
         // Ouverture de l'URL pour une lecture
         uneUrl = new URL(args[0]);
         //Convertir l'URL en InputStream
        fluxE=uneUrl.openStream() ;
        //Construction du flux de type BufferedInputStream
        donneesLues = new BufferedInputStream(fluxE);
         // construction d'un FileOutputStream.
         fluxS = new FileOutputStream(args[1]);
         // construction d'un BufferedOutputStream
         donneesEcrites = new BufferedOutputStream(fluxS);
         System.out.println("Connextion établie") ;

         tampon=new byte[TAILLE_TAMPON] ;
         do{
             nbLus=donneesLues.read(tampon) ;
             if (nbLus !=-1) {
                  donneesEcrites.write(tampon,0,nbLus) ;}
            }while(nbLus !=-1) ;
          donneesEcrites.close() ;     //écriture de la fin du fichier
          System.out.println("Fin d'écriture") ;
      }
      catch (MalformedURLException excp) {
         System.out.println (args[0] + " : impossible de traiter cette URL");
         System.out.println (excp);
      }
      catch (IOException excp) {
         System.out.println ("Erreur : " + excp);
      }
   }
}
```

Tester cette classe en créant un fichier URL2Fichier.java.

Exercice 2 :

On veut créer une classe qui copie un fichier texte source vers un fichier texte destination ligne par ligne. Compléter le programme de cette application dont le squelette est donné ci-dessous :

```
public class CopieFichierTexte {
        private String source;
        private String destination;
        public CopieFichierTexte(...) {
                ...
        }
        public static void main(String[] args) {
                try {
                        ...
                }
                catch (...) {
                        System.out.println("erreur à l'ouverture des flux");
                }
                catch (...) {
                        System.out.println("erreur lors des lectures/écritures");
                }
        }
        public void copieLignes() throws ... {
                ...
        }
}
```

TD 5 : Applications client-serveur

Exercice 1 : Création d'un serveur TCP/IP mono-client

On veut créer un serveur de temps simple (par sockets TCP-IP) à partir de l'exemple de l'application serveur étudiée en cours. Modifier le programme de façon à ce que le serveur retourne l'heure et la date du système. On utilisera ici le port 6666 comme port d'écoute du serveur. Pour l'acquisition de la date et de l'heure, on utilisera une instance de Date :

```
Date d=new Date() ;    //d contient la date et l'heure au moment de son instantiation
String s=d.toString() ; //s contient la date et l'heure sous la forme d'un String
```

Utiliser un client Telnet pour tester cette application.

Exercice 2 : Création d'un client TCP/IP

Ecrire un programme client permettant de lire la date sur un serveur de date.

Exercice 3: Synthèse

1. On veut créer un client simple en java permettant de se connecter à un serveur pour lire un fichier texte qui a pour URL : http://www.u-pec.fr/index.html. Chaque ligne lue est ensuite affichée à l'écran (côté poste client).

2. Modifier l'application précédente pour que cette fois-ci chaque ligne lue par le client soit envoyée vers un serveur d'adresse 194.214.10. 245 écoutant sur le port TCP 80.