Guide de révision pour l'examen

Redouane EZZAHIR
red.ezzahir@gmail.com

Avant propos

- Le contenu de ces diapositives n'est pas un cours, mais une moyenne pour vous aider à réviser votre cours java et de bien se préparer à l'examen.
  - Les points principaux à réviser
- Référenciez-vous aux supports de cours et aux liens utiles suivants en cas de besoin
  - https://sites.google.com/site/redezzahir/teaching
  - http://blog.paumard.org/cours/java/
  - http://cs.armstrong.edu/liang/intro9e/javasupplement.html
- Marquez les points non claires pour les discuter en classe

Faire une recherche web ou bibliographique sur

- Une peut d'historique de java
- Les commandes de java (comment compiler, déboguer et exécuter un programme)
- Consultation en ligne de la documentation Java
- Notion de packages
- Notion de variables
- Notion de types
- Les opérateurs
- conversion (casting)
- Conversions de type (String to int : Integer.parseInt(s))

Les commentaires java

- Java permet un commentaire en ligne - tout ce qui est sur la ligne après // est un commentaire.
- Java supporte les commentaires de bloc - le commentaire commence par / * , se termine par * / , et comprend toutes les lignes entre.
- Java prend en charge également les commentaires entre /** et **/ .
- Ce sont comme des blocs de commentaires, mais peuvent être utilisés pour générer automatiquement la documentation avec javadoc .

Operateurs

- Les expressions booléennes employant les opérateurs ET logique (&&) et OU logique (||) sont évaluées de manière paresseuse
  - L'évaluation s'arrête aussitôt que le résultat est déterminé.

- L'opérateur & fonctionne exactement comme l'opérateur &&, et l'opérateur | fonctionne exactement comme l'opérateur || avec une exception: les opérateurs & et | évaluent toujours à la fois opérandes.

- <<= (>>=) Décaler à gauche (droite) et affectation sur (byte char short int

- >>>= Décaler à droite (signe aussi) et affectation

Structure de contrôle et boucles

if else, for, while, do…while, switch

- Syntaxe similaire à la syntaxe du C++;
- switch (expr) { // expr ne pet être que int ou char

```
case valeur:
instructions;
break;  // sortir du bloque
...
default:
instructions;
}
```

- continue : dans une boucle, sauter le este du bloque et passer à l'itération suivante;

Les tableaux

Déclaration : type[ ] identificateur ;

Construction : new type[taille]

Initialisation avec éléments connus :

type[ ] t1D ={val_1, ... , val_{taille}} ;

type[ ][ ] t2D = {{val_1, ... , val_{taille}} ,
{val_1, ... , val_{taille}} } ;

Traitement des arguments de la méthode main

```
public static void main(String args[]) {
... }
```

```
java MonProgramme texte1 127 unfichier.txt
```

Le tableau args contiendra les élément suivants :

args[0] : la chaîne de caractères "texte1"
args[1] : la chaîne de caractères "127"
args[2] : la chaîne de caractères "unfichier.txt »

Notation à point

- C’est la syntaxe utilisée pour référencier des valeurs et des méthodes à l'intérieur d'un objet ou d'une classe.

- Par exemple, m.color = "jaune"; affectation de la variable d'instance **color** de l’objet **m** à la valeur string **jaune** .

- Nous avons utilisé la notation à points pour appeler des méthodes:
  - System.out.println ("Salut");

Les chaînes de caractères

Déclaration/initialisation

String s1 = "Bonjour" ;
String s2 = "tous le monde"

Concaténation String s = s1 + s2

String est immutable

Accès au (i+1)-ième caractère : chaine.charAt(i) ;

chaine.length() : retourne la taille de la chaine
chaine.equals(position, chaine2) : comparaison alphabétique :
chaine1.compareTo(chaine2): comparaison lexicographique
chaine.replace(char1,char2) : remplacement de caractère mais chaine n'est pas modifiée un nouveau objet est retourné.
chaine.indexOf(char): recherche d'une caractère
chaine.substring(indice1, indice2): extraction de sous-chaine

Opérateurs, de l'égalité et String

- Java ne supporte pas la surcharge d'opérateur, vous devez utiliser des méthodes à sa place.
- En général, les opérateurs fonctionnent uniquement avec les types primitifs, ou de la classe String.
- Égalité (== et ! =) est une exception, mais ils ne fonctionnent pas comme vous vous en doutez!

```
String s1 = new String ("Cathy");
String s2; s2 = s1; // même endroit dans la mémoire
if(s1 == s2) // true
   ...
s2 = new String (s1); // ont chacun leur propre mémoire
if(s1 == s2) // faux, et probablement faux!
...
 if (s1.equals(s2)) // true, et probablement ce que
…                   //  vous voulez vraiment!
```

Programmation orientée objet

- Deux concepts fondamentaux:
  - *Abstraction* : Le processus d'abstraction consiste à identifier pour un ensemble d'éléments :
    - des caractéristiques communes à tous les éléments
    - des mécanismes communs à tous les éléments
    - ☞ description générique de l'ensemble considéré : se focaliser sur l'essentiel, cacher les détails.
    - ☞ Notion de classe en java
  - *Encapsulation En plus de l'abstraction l'encapsulation permet de définir deux niveaux de perception :*
    - ***Externe** : visible par les programmeurs-utilisateurs de l'objet*
    - ***Interne** : :détails d'implémentation de l'objet*
    - ☞ Notion de visibilité en java

Encapsulation

- L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure (class) en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés.

- L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

- **Principe fondamental** : la modification d'une instance ne peut se faire que via son interface (ses méthodes publiques)

Quiz

- Pouvons nous réellement créer un objet EncapTest?

```
public class EncapTest{
        private String name;
        private String idNum;
        private int age;
}
```

Quiz

- Pouvons nous réellement créer un objet EncapTest

```
public class EncapTest{
        private String name;
        private String idNum;
        private int age;
}
```

- Non, il faut un constructeur et des méthodes d'accès

Encapsulation (Exemple)

```
public class EncapTest{
        private String name;
        private String idNum;
        private int age;
        EncapTest(String name, String idNum){
         super(); this.name = name; this.idNum = idNum; }
        public int getAge(){ return age; }
        public String getName(){ return name; }
        public String getIdNum(){ return idNum; }
        public void setAge( int newAge)throws NegatifAgeException{
                if(age<=0) throw new NegatifAgeException();
                age = newAge;
        }
        public void setName(String newName){
                name = newName;
        }
        public void setIdNum( String newId){
         idNum = newId;
        }
}
```

Faille d'encapsulation (Exemple)

```
class Horaire {
  private String jour;
  private double heure;
  public Horaire(String j, double h){
  // contrôle que le jour et
  //l'heure sont ok de manière générale
        jour = j;
        heure = h;
  }
  public void setHeure(double h) {
        heure = h; }
}
```

```
class Examen {

private String Nom;
private Horaire heure ;

public String getNom() {
        return nom;
}
public Horaire getHoraire() {
        return heure ;
}
}
```

Faille d'encapsulation (Exemple)

```
class Horaire {
  private String jour;
  private double heure;
  public Horaire(String j, double h){
  // contrôle que le jour et
  //l'heure sont ok de manière générale
        jour = j;
        heure = h;
  }
  public void setHeure(double h) {
        heure = h; }
}
```

```
class Examen {

private String Nom;
private Horaire heure ;

public String getNom() {
        return nom;
}
public Horaire getHoraire() {
        return heure ;
}
}
```

```
Examen ds1 = new Examen(« Informatique");
ds1.setHoraire(« Mardi", 14.0); // valeurs ok
// pour un examen
Horaire hor = ds1.getHoraire(); //SOURCE DU PROBLEME ICI
// ...
// ah il y a une séance de cinema le mardi à 21h
hor.setHeure(21); // OK pour un horaire général
// ...
// j'affiche l'horaire de mon DS:
System.out.println(ds1.getHoraire());
//Aou!.. un test le mardi 21h !!
```

Faille d'encapsulation (Solution)

Solution : faire en sorte que getHoraire retourne une référence à une copie de l'horaire de l'examen

```
class Examen {
//....
public Horaire getHoraire() {
// prévoir un constructeur de copie
// dans la classe Horaire:
    return new Horaire(horaire);
  }
}
```

Exercice

Trouvez la faille dans cette façon de définir ce second setter pour la classe Examen de l'exemple précédent :

```
class Examen {
//....
public void setHoraire(Horaire unHoraire) {
 // contrôle que unHoraire est ok
 // pour un examen, et si oui:
   horaire = unHoraire;
 }
}
```

Exercice (solution)

- la méthode setHoraire ne fait qu'une simple copie de la référence de l'objet en paramètre (copie surface). Toute modification sur la référence du paramètre unHoraire se répercutera sur la donnée horaire encapsulée dans l'objet Examen.

- Le code correcte est :

```
class Examen {
//....
public void setHoraire(Horaire unHoraire) {
 // copie profond de l'objet Horaire
   horaire = new Horaire(unHoraire);
 }
}
```

Variable de classe vs variable d'instance

- Une variable/méthode de classe est une valeur/fonction dans une classe qui est commune à toutes les instances de cette classe

- Une variable d'instance stocke les valeurs qui sont liées à un objet. La création d'une variable d'instance suit la même syntaxe de base que la déclaration d'une variable locale, mais déclarée à l'intérieur de la classe

Méthodes de classe vs Méthodes d'instances

- Une méthode de classe est une méthode static

```
Class UneClasse{
     static type nomMethode(type1 arg1, ..., typeN argN){
    //corps
    }
}
```

- Méthodes d'instance toute méthode non-static

```
Class UneClasse{
    type nomMethode(type1 arg1, ..., typeN argN){
    //corps
    }
}
```

Variables de classe vs Variables d'instances

- Une variable de classe est une variable static

```
Class UneClasse{
     static type nomVariable;
}
```

- Une variable d'instance est une variable non-static

```
Class UneClasse{
        type nomVariable;
}
```

Accès aux membres static

- L'accès aux membre static se fait via le nom de la classe (NomClasse.nomMembre).

```
Class  ClassA{
    static  int var = 10;
    static  int carre(int v){
       return v*v;
     }…
   }
```

```
Class ClassB{
     public static void main(String[] arg){
              System.out.println("le carré de " +
   ClassA.var + " est :" + ClassA.carre(ClassA.var ));}}
```

Accès aux membres non-static

- L'accès aux membre d'instance (non-static) se fait via le nom de l'instance (nomDinstance.nomMembre.

```
Class  ClassA{
     public int var = 10;
     public int carre(int v){
       return v*v;
     }…
   }
```

```
Class ClassB{
     public static void main(String[] arg){
ClassA a = new ClassA(); // creation de l'instance
                System.out.println("le carré de " +    a.var
   + " est :" + a.carre(ClassA.var ));}}
```

Accès aux membres static/non-static

- Il est possible d'accéder à un membre statique via le nom d'une instance (non recommandé)

- Dans un bloque static, on ne peut accéder directement que aux membres static.

Notion de signature

- Ce qui correspond à un en-tête de fonction

```
1. public static int add2ints (int a, int b)
2. public static int add2ints (double a, double b)
3. public static int add2ints(double a, int b)
4. public static int add2ints (int a, double b)
```

- Quelle méthode est acceptée par le compilateur ou appelée dans la main?

```
answer = add2ints (123, 8,7);
answer = add2ints (11.5, 21.3);
answer = add2ints (18, 23);
```

- Ceci est appelé surcharge de méthode

Surcharge et redéfinition

- Surcharge de méthode

```
Class A {
 void affiche (int arg) {…}
 void affiche (double arg) {…}
 void affiche (int arg1, int arg2) {..}
..
}
```

- Redéfinition de méthode

```
Class B extends A {
 void affiche (int arg) {
   ..
 }
}
```

Surcharge et redéfinition

- Les méthodes de classes et les méthodes d'instances peuvent être surchargées et redéfinies
- Une méthode de classe (static) ne peut pas être redéfinie en méthode d'instance et vice-versa.

```
Class A {
 void affiche (int arg) {…}
}
```

```
Class B extends A {
 static void affiche (int arg) {…}
}
```

Passage par valeur/référence

- Type primitive → passage par valeur

  int carre(int a) {a*=a; return a};

  Int x =1; carre(x); // carre ne peut modifier la valeur de x;

- Passage d'une référence (par valeur aussi) :
  - type m(typeObjet arg) ;
    - m(**v**) → la référence **v** ne peut pas être modifiée par m
    - m(v) → l'objet référencé par v peut être modifié par m

Masquage : un argument caché un attribut de même nom

```
int  largeur, longueur;
rectangle(int largeur, int longueur){
```

Solution (this) :

```
    this.largeur=largeur;
    this.longueur= longueur;
}
```

Le modificateur final

- Le modificateur final peut être associée à une variable, une méthode ou une classe :

- variable finale : ne peut être modifiée après initialisation (constante)
- méthode finale : ne peut être cachée par redéfinition.
- classe finale : ne peut avoir de sous-classes

Héritage

- L'héritage est l'un des concepts les plus importants de la POO.
- L'héritage est un mécanisme très puissant pour la réutilisation du code et est une caractéristique importante de tous les langages de programmation orientés objet.
- Les classes sont organisées dans une hiérarchie. La classe en haut est appelée superclasse et les classes inférieures sont ses sous-classes .

Héritage en java

Pour spécifier un lien d'héritage :

class Sousclasse extends SuperClass {...}

Masquage : un attribut/méthode peut être redéfini (caché) dans une sous-classe

Accès à un membre caché : super.membre

Le constructeur d'une sous classe doit faire appel au constructeur de la super classe :

```
class SousClasse extends SuperClasse {
  SousClasse(liste de paramètres) {
   super(arguments) ; // première instruction
  …}
}
```

La classe Object

- Toute classe dérive implicitement de la classe Object.

```
public class Object {
  public String toString() { ... }
  public final Class getClass() { ... }
  public boolean equals(Object o) { ... }
  public int hashCode() { ... }
  protected Object clone()
    throws CloneNotSupportedException { ... }
  ...
}
```

- Les deux définitions suivantes sont équivalentes :
  - class A { ... }
  - class A extends Object { ... }
- Les méthodes de la cette classe sont souvent redéfinies dans les sous classes.

Méthode et classes abstraites

- **Méthode abstraite :**
  - abstract Type nomMethode(liste d'arguments) ; // pas de corps
- **Classe abstraite** : abstract classeAbstraite{..}
- Une classe contenant une méthode abstraite doit être déclarée abstraite.
- L'idée est que la classe abstraite est utilisée pour organiser la hiérarchie, mais n'est pas destiné à être instancié.

Interface

- La notion d'héritage multiple n'existe pas en JAVA.
- Mais on peut cependant utiliser la notion d'interface pour attribuer des composants communs à des classes non-liées par une relation d'héritage
  - Exemple: Comparable, Cloneable, Serializable

```
interface Interface{
    Type methode1(Type1 arg1...);
    Type methode2(Type1 arg1...);
        . . .
}
```

Par défaut
public et abstarct

- Les interfaces permettent de concevoir une hiérarchie qui peut être étendue aux classes qui n'ont pas encore été créés.

Polymorphisme (1)
Résolution statique/ dynamique

- Résolution statique choix de la variable lors de l'exécution du programme en fonction de type de la référence.

- Résolution dynamique des liens : choix des méthodes à invoquer lors de l'exécution du programme en fonction de la nature réelle des instances

Polymorphisme (2)

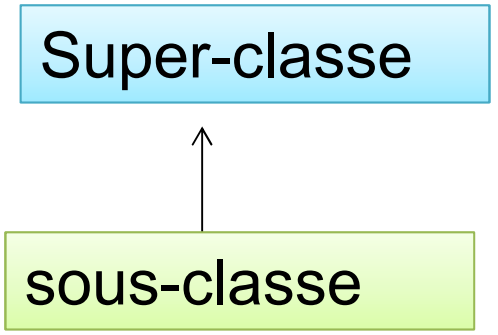
Résolution statique/ dynamique (Exemple)

```
class A{
int a = 1;
char m(){
return 'A';
}
}
```

```
class B extends A{
    int a = -1;
    int b;
    char m(){
    return 'B';
    }
}
```

```
B x  = new B();
A y = x;
System.out.println(y.m() + " "+ y.a); // B 1
```

Polymorphisme (2) sous-typage



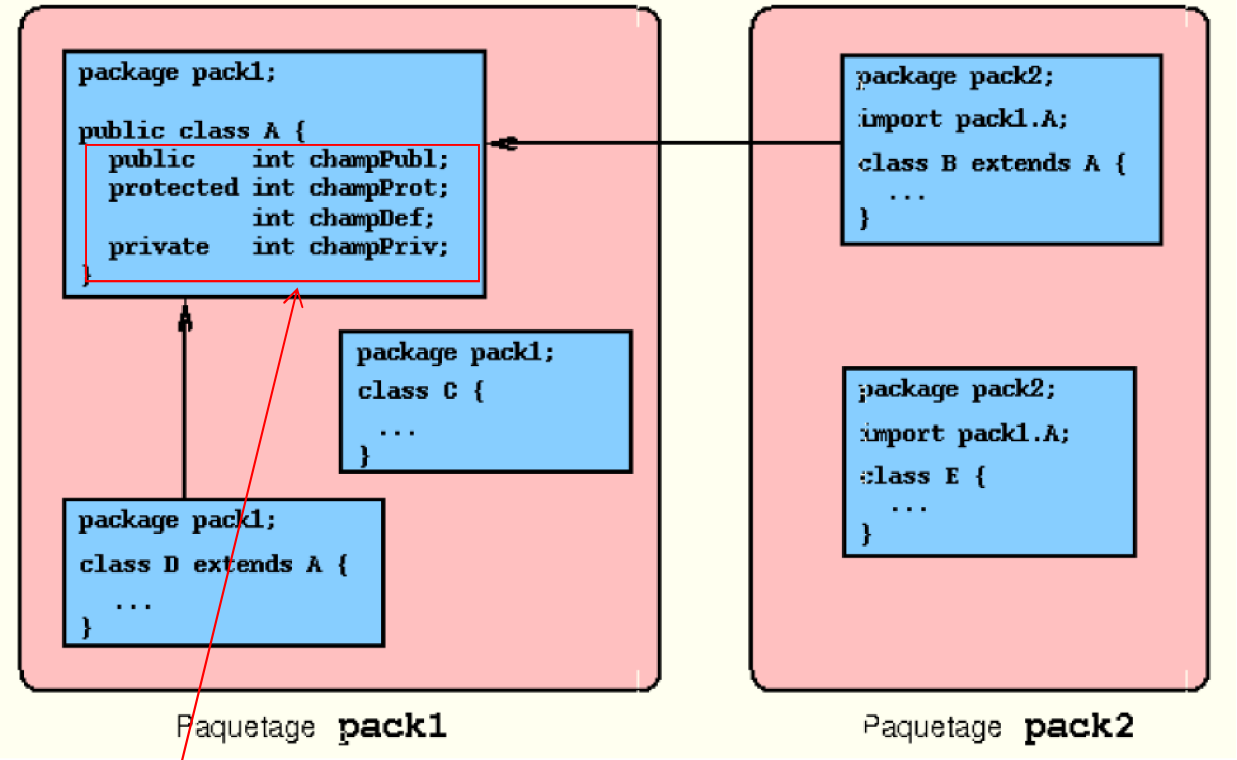
- Une référence de sous-classe peut être affecté à une référence de super-classe

```
class A implements IA{
…
}
class B extends A{
}
```

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

a = b; //ok
IA c = a; //ok
IA d = b; //ok
b=a ; // Error
b= (B) b // ok
```

La visibilité de membres



Visible Dans	A	B	C	D	E
champPub	oui	oui	oui	oui	oui
champProt	oui	oui	oui	oui	non
champDef	oui	non	oui	oui	non
champPriv	oui	non	non	non	non

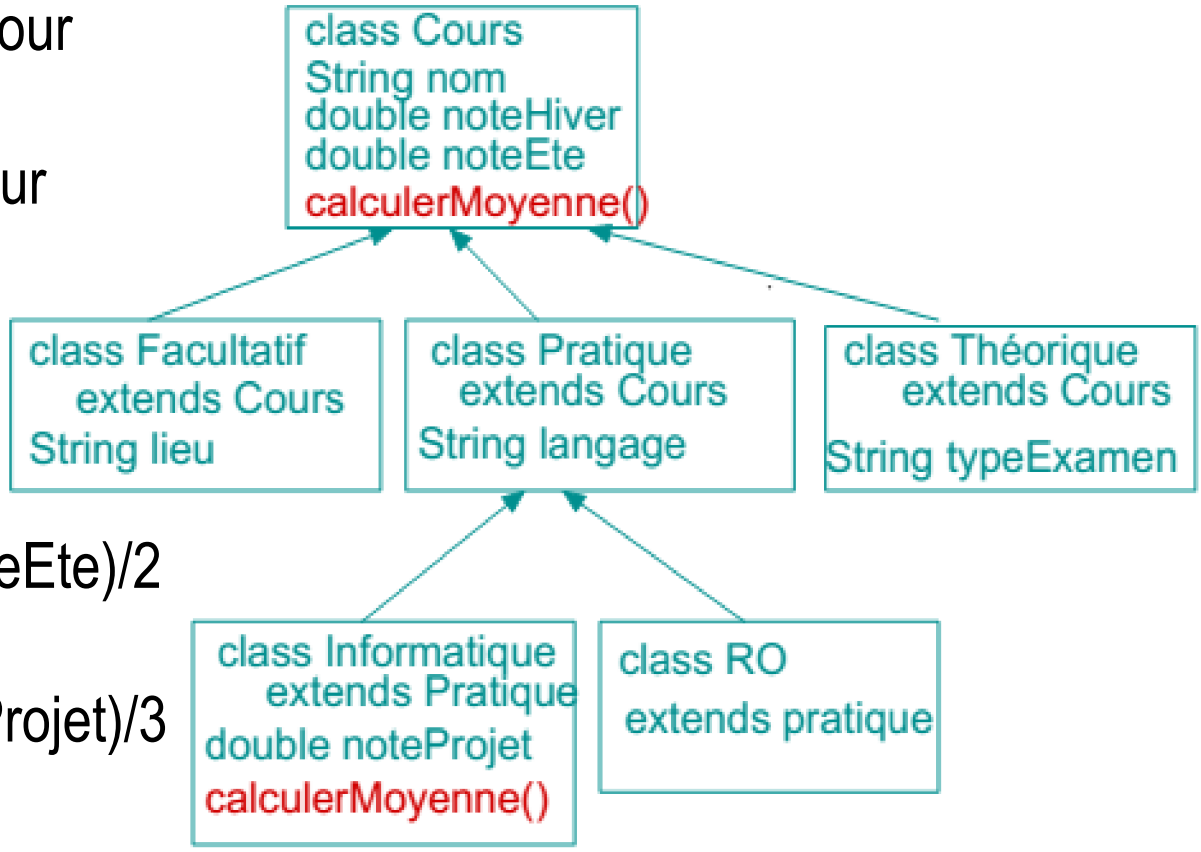
Exercice

En se basant sur l'hiérarchie de classes ci-dessous, écrire un programme qui mis en ouvre les principes de la POO et qui calcule la moyenne générale pour un étudiant. Utiliser :

-Un tableau de type Cours pour stocké les 4 cours.

-L'étudiant saisi les notes pour les quatre cours.

-Affichage de la moyenne



-Moyenne =(noteHiver +noteEte)/2

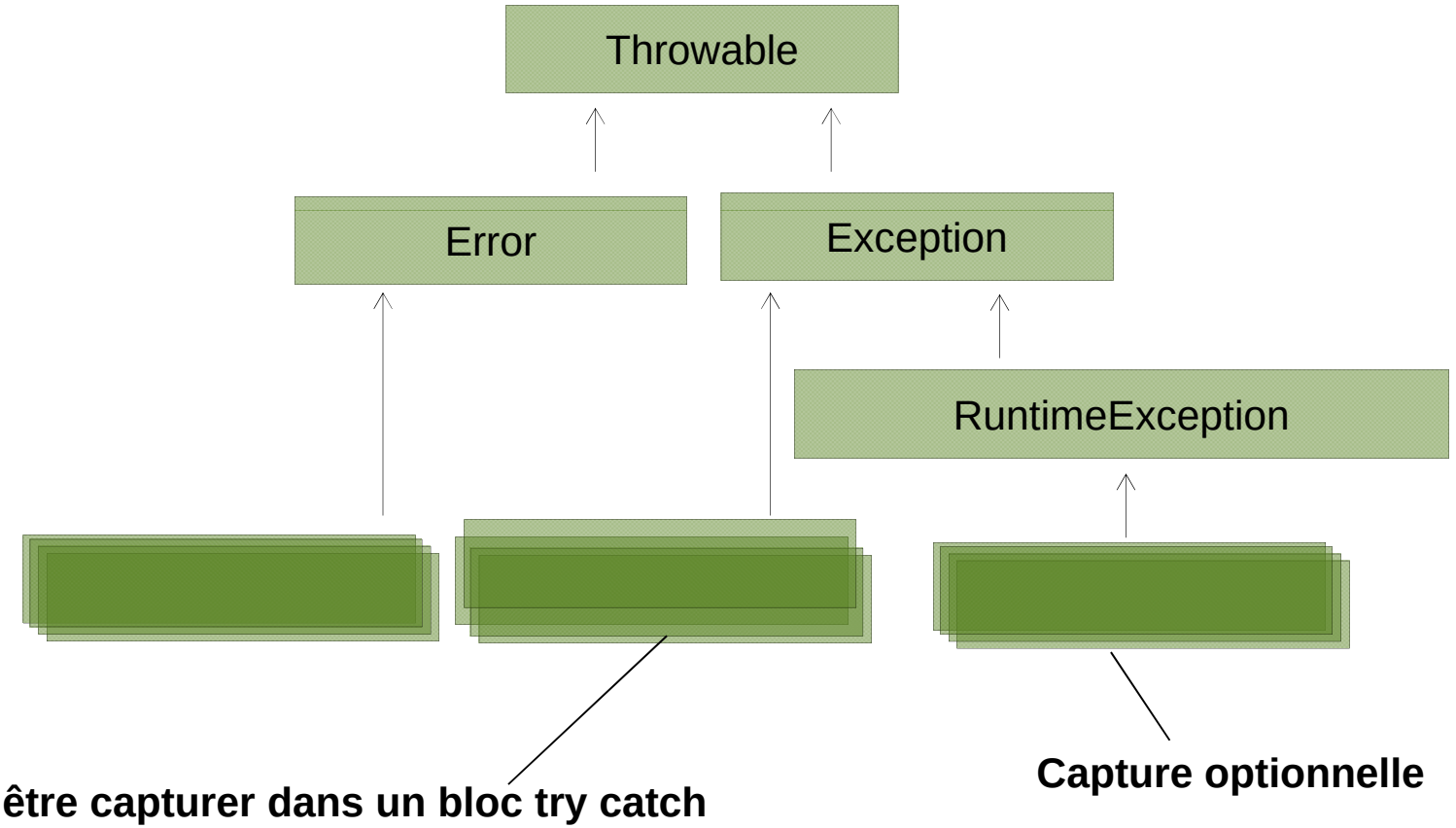
-Sauf pour informatique

(noteHiver +noteEte+ noteProjet)/3

Types d'exceptions

Il existe 3 trois types d'exceptions organisés comme ceci :

Arbre de sous-typage des exceptions



Exception Mots clés

- throw indique l'erreur (i.e. « lance » l'exception)
- try indique un bloc réceptif aux erreurs
- catch gère les erreurs associées (i.e. les « intercepte » pour les traiter)
- finally (optionel) indique ce qu'il faut faire après un bloc réceptif.
- throws : méthode lançant une exception sans la traiter localement doit généralement informer qu'elle le fait Ceci se fait en ajoutant une clause throws à l'entête de la méthode

Exception (Exemple)

- class DataFormaException extends exception{...}
- decrypt(InputStream in, OutputStream out) throws IOException{...}
- verify(InputStream in)throws DataFormaException, IOException{…}

```
public static void main(String[] args) throws IOException{
  InputStream inCrpt =new FileInputStream(args[0]);
  try {
       OutputStream outDecrpt=new FileOutputStream(arg[1]);
      try {
        decrypt(inCrpt , outDecrpt);// decriptage de fichier
        outDecrpt.close();
        if(verify(new FileInputStream(args[1]))){
              //fait qqc
        }
      }catch(DataFormaException e){
              System.err.println("Erreur de Formatage de donnees");
      } finally {
       out.close();
      }
   }
   finally {
     in.close();
   }
}
```

Quiz

	Vrai	Faux
• Les blocs **try** doivent avoir au moins un bloc catch	☐	☐
• Les blocs **try** doivent avoir un bloc finally	☐	☐
• Le bloc finally est obligatoire	☐	☐
• Lors d'une exception dans un **try**		
◦ Le reste du bloc **try** est sauté	☐	☐
◦ Le catch idoine est exécuté, s'il est présent	☐	☐
◦ Le catch idoine est sauté si **finally** est présent	☐	☐
◦ Le **finally** est exécuté, s'il est présent	☐	☐
• Lorsqu'un **try** ne déclenche pas d'exception		
◦ Le **finally** est sauté, s'il est présent	☐	☐

Quiz

	Vrai	Faux
• Les blocs **try** doivent avoir au moins un bloc catch	☐	☐
• Les blocs **try** doivent avoir un bloc finally	☐	☐
• Le bloc finally est obligatoire	☐	☐
• Lors d'une exception dans un **try**		
◦ Le reste du bloc **try** est sauté	☐	☐
◦ Le catch idoine est exécuté, s'il est présent	☐	☐
◦ Le catch idoine est sauté si **finally** est présent	☐	☐
◦ Le **finally** est exécuté, s'il est présent	☐	☐
• Lorsqu'un **try** ne déclenche pas d'exception		
◦ Le **finally** est sauté, s'il est présent	☐	☐

Quiz

- Quel est le problème dans ce code ?

```
int numElems;
public void add(Object o) throws SomeException
{
....
numElems++;
if (maListe.maxElem() < numElems)
{
// réallouer, recopier, risque d'exceptions
}
maListe.addToList(o); //risque d'exception
}
```

Quiz

- Quel est le problème dans ce code ?

```
int numElems;
public void add(Object o) throws SomeException
{
....
numElems++;
if (maListe.maxElem() < numElems)
{
// réallouer, recopier, risque d'exceptions
}
maListe.addToList(o); //risque d'exception
}
```

- Réponse : L'exception peut survenir après l'augmentation de la taille du conteneur : L'objet n'est pas laissé dans un état valide…

Quiz (vrai/Faux)

- RuntimeException doit être obligatoirement captée par try/catch
- Les assertions sont toujours exécutées.
- IOException correspond aux erreurs d'entré/sortie
- MyException extends Exception est de type Throwable
- Il est possible d'utiliser try/catch dans un bloc catch.

Quiz (vrai/Faux)

- RuntimeException doit être obligatoirement captée par try/catch
- Les assertions sont toujours exécutées.
- IOException correspond aux erreurs d'entré/sortie
- MyException extends Exception est de type Throwable
- Il est possible d'utiliser try/catch dans un bloc catch.

Compiler le code

```
public class TestException {
    public static void main(java.lang.String[] args) {
        // Insert code to start the application here.
        int i = 3;
        int j = 0;
        try {
            System.out.println("résultat = " + (i / j));
        }
        catch (Exception e) {
        }
        catch (ArithmeticException e) {
        }
    }
}
```

Compiler le code (solution)

```
public class TestException {
    public static void main(java.lang.String[] args) {
        // Insert code to start the application here.
        int i = 3;
        int j = 0;
        try {
            System.out.println("résultat = " + (i / j));
        }
        catch (Exception e) {
        }
        catch (ArithmeticException e) {
        }
    }
}
```

Ce bloc ne sera jamais atteint ArithmeticException est un sous type de Exception. Il faut switché les deux blocs catch

Boucles étiquetées (labeled loops)

- Java permet les boucles marquées qui vous permettent de sortir de boucles multiples imbriquées en utilisant l'étiquette d'une instruction break.

```
int[] t= new int[]{1,8,3,5};
PLUSGRANG:

for (int i = 0; i < t.length; i++) {
        for (int j = 0; j < t.length; j++){
                if ( t[i] > t[j] )
                        break PLUSGRANG;
                else if ( t[i] < t[j] )
                        break;
        }
        System.out.println("le break va finir ici!");
}
System.out.println("le break PLUSGRANG finira ici!");
```

- Notez que, techniquement, ce n'est pas un goto - Java ne supporte pas gotos.

Exercice (boucles for, while, do ... while)

- Quelle boucle utiliserez-vous dans chacune des cas suivant?
  - Demander à l'utilisateur jusqu'à ce qu'ils appuyez sur Entrée ...
  - Demandez à l'utilisateur combien de copies qu'ils veulent et ensuite imprimer ce nombre.
  - Parcourez la liste jusqu'à ce que vous atteignez la fin et faites quelque chose pour chacun.
  - Parcourez la liste de N éléments et faire quelque chose pour chacun.
  - Invite l'utilisateur à entrer un nom de fichier et si elles entrent "stop", quittez le programme.
  - Demander à l'utilisateur le nombre d‘étudiants, puis lui demander de saisir le nom de chacun d'eux.
  - Parcourez la liste des étudiants en cherchant le nom "Ali"

Exercice (boucles for, while, do ... while)

- Quelle boucle utiliserez-vous dans chacune des cas suivant?
  - Demander à l'utilisateur jusqu'à ce qu'ils appuyez sur Entrée ...
  - Demandez à l'utilisateur combien de copies qu'il veuille et ensuite imprimer ce nombre.
  - Parcourez la liste jusqu'à ce que vous atteignez la fin et faites quelque chose pour chacun.
  - Parcourez la liste de N éléments et faire quelque chose pour chacun.
  - Invite l'utilisateur à entrer un nom de fichier et si elles entrent "stop", quittez le programme.
  - Demander à l'utilisateur le nombre d'‘étudiants, puis lui demander de saisir le nom de chacun d'eux.
  - Parcourez la liste des étudiants en cherchant le nom "Ali"

La gestion de fichiers

La gestion de fichiers proprement dite se fait par l'intermédiaire de la classe **java.io.File**

Un objet de la classe **File** représente un **chemin textuel** vers un fichier ou un répertoire.

File (String name)
File (String path, String name)
File (File dir, String name)
boolean isFile( ) / boolean isDirectory( )
boolean mkdir( )
boolean exists( )
boolean delete( )
boolean canWrite( ) / boolean canRead( )
File getParentFile( )
long listFiles( )
…

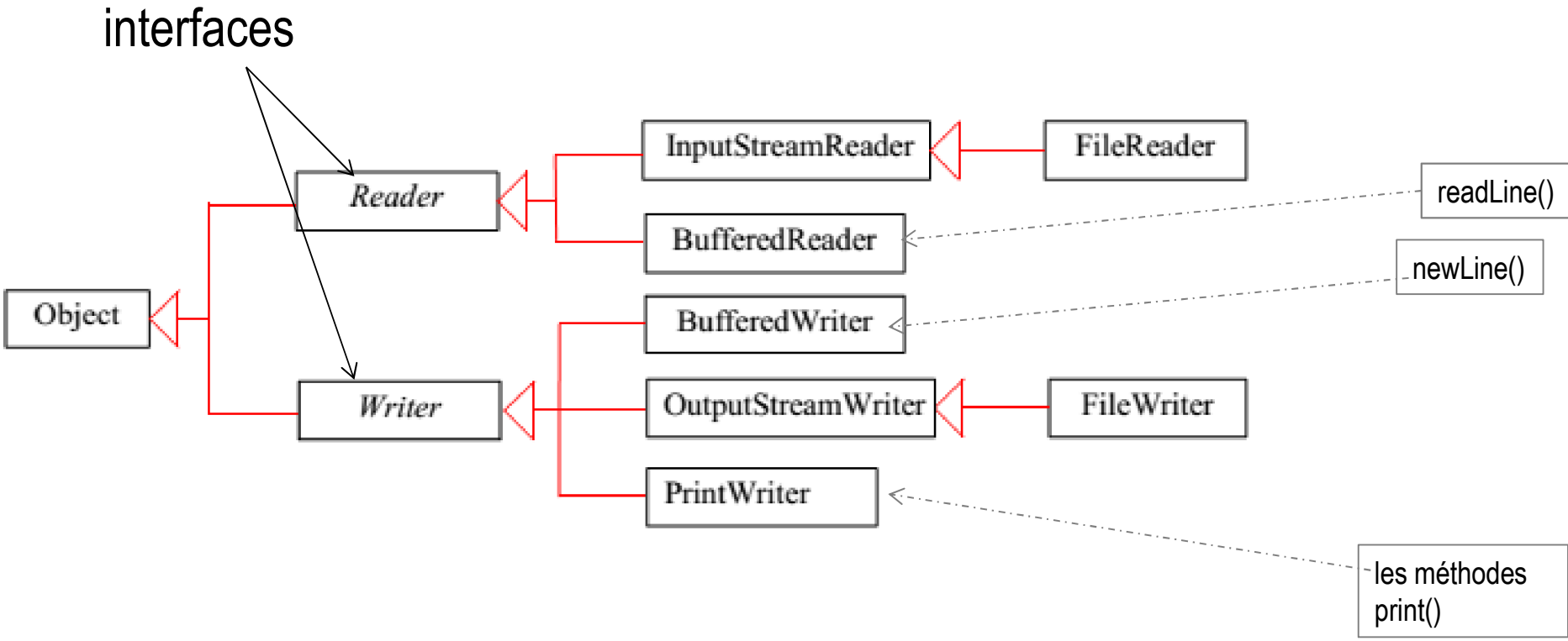
```
public static void main(String[] args) {
    FilenameFilter txtFilter=new FilenameFilter() {
     public boolean accept(File dir, String name) {
        return name.endsWith(".txt");
      }
    };
    File file=new File(".");
    System.out.println(Arrays.toString(
      file.listFiles(txtFilter)));

     FileFilter dirFilter=new FileFilter() {
     public boolean accept(File file) {
        return file.isDirectory();
      }
    };
    System.out.println(Arrays.toString(
                        file.listFiles(dirFilter)));
  }
```

E/S texte (caractère)

- En JDK 1.5, pour manipuler l'E/S texte on utilise les classes Scanner et PrintWriter. Cela simplifie grandement l'E/S texte
  - Scanner in = new Scanner(fileName)
    - next(), nextInt(), `hasNextLine(), nextLine()` …
  - PrintWriter out = new PrintWriter (fileName)
    - print(), println() , …
- Pour les enceintes versions : Reader/Writer, et leur extensions

Flux de caractère (Reader/Writer)



InputStreamReader/OutputStreamWrite sont utilisées pour convertir entre les octets et de caractères.

PrintStream offre les mêmes fonctionnalités que **PrintWriter** mais moins performant → remplacé par PrintWriter dans JDK 1.2

Flux de caractère

Reader

- Flux de **char** en entrée (méthode bloquante)
  - Lit un char et renvoie celui-ci ou -1 si c'est la fin du flux
    - abstract int **read**()
  - Lit un tableau de char (plus efficace) du flux vers le tableau b
    - int **read**(char[] b)
    - int **read**(char[] b, int off, int len)
  - Saute un nombre de caractères
    - long **skip**(long n)
  - Ferme le flux
    - void **close**()

Writer

- Flux de caractère en sortie
  - Ecrit un caractère, un int pour qu'il marche avec le read
    - abstract void **write**(int c)
  - Ecrit un tableau de caractère (plus efficace) du tableau b vers le flux
    - void **write**(char[] b)
    - void **write**(char[] b, int off, int len)
  - Demande d'écrire ce qu'il y a dans le buffer
    - void **flush**()
  - Ferme le flux
    - void **close**()

Copie de flux

- Caractère par caractère (mal)

```
public static void copy(Reader in,Writer out)
  throws IOException {

  int c;
  while((c=in.read())!=-1)
    out.write(c);
}
```

- Par buffer de caractère

```
public static void copy(Reader in,Writer out)
  throws IOException {

  char[] buffer=new char[8192];
  int size;
  while((size=in.read(buffer))!=-1)
    out.write(buffer,0,size);
}
```

FileReader/FileWriter

- Permet de créer un Reader sur un fichier
  - **FileReader**(File file)

  Renvoie l'exception **FileNotFoundException** qui hérite de **IOException** si le fichier n'existe pas

  L'encodage utilisé est celui de la plateforme
- Permet de créer un Writer sur un fichier
  - **FileWriter**(File name, boolean append)

  append à true si l'on veut écrire à la fin

FileReader/FileWriter (Exemple)

```
import java.io.*;

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

 FileReader in = new FileReader(inputFile);
 FileWriter out = new FileWriter(outputFile);
 int c;

 while ((c = in.read()) != -1)
 out.write(c);

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

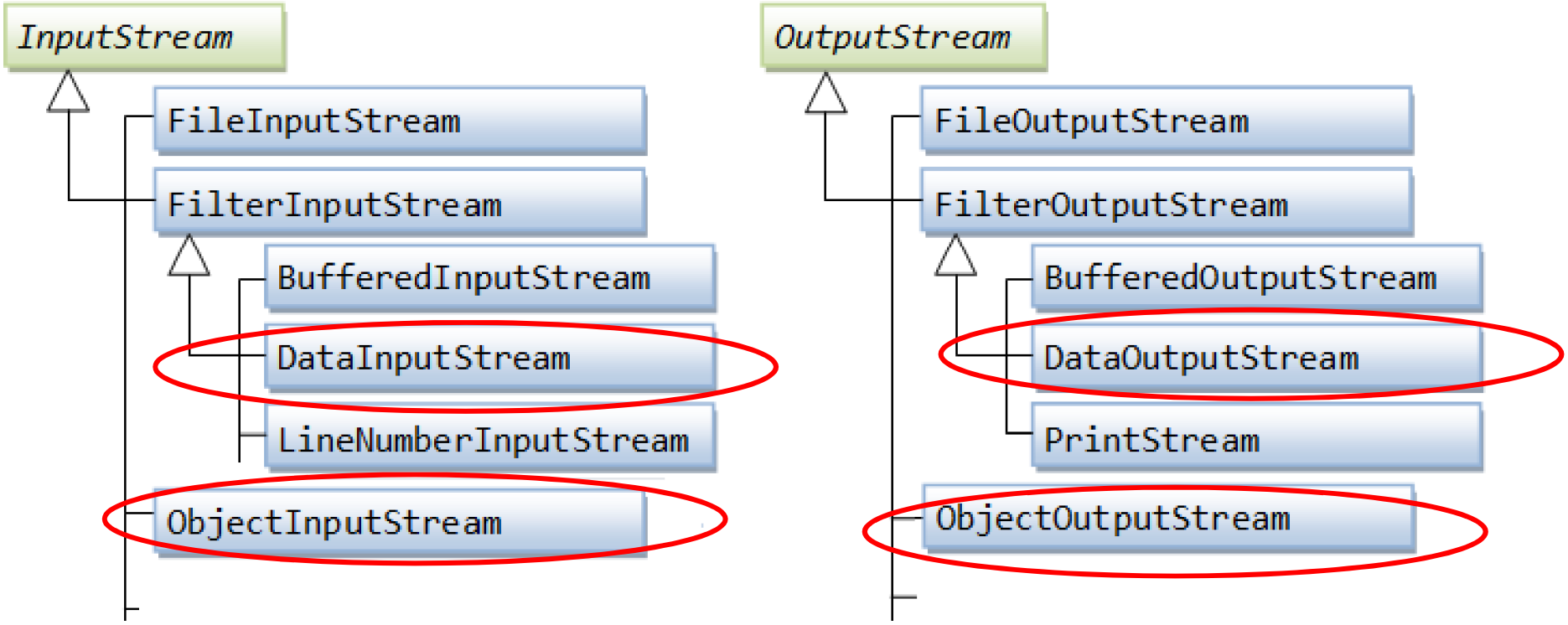
File{Reader/Writer} et encodage

- Un File{Reader/Writer} utilise toujours l'encodage de la plateforme si l'on veux utiliser un autre encodage, il faut utiliser un InputStreamReader ou OutputStreamWriter

```
public static void main(String[] args) throws ... {
  File fileIn=new File(args[0]);
  File fileOut=new File(args[1]);
  Charset charsetIn=Charset.forName(args[2]);
  Charset charsetOut=Charset.forName(args[3]);

  InputStreamReader reader=new InputStreamReader(
    new FileInputStream(fileIn),charsetIn);
  OutputStreamWriter writer=new OutputStreamWriter(
    new FileOutputStream(fileOut),charsetOut);
  copy(reader,writer);
}
```

Flux par byte



Flux par byte

InputStream

- Flux de **byte** en entrée
  - Lit un byte et renvoie ce byte ou -1 si c'est la fin du flux
    - abstract int **read**()
  - Lit un tableau de byte (plus efficace) du flux vers le tableau b
    - int **read**(byte[] b)
    - int **read**(byte[] b, int off, int len)
  - Saute un nombre de bytes
    - long **skip**(long n)
  - Ferme le flux
    - void **close**()

OutputStream

- Flux de **byte** en sortie
  - Ecrit un byte, on utilise un int pour qu'il marche avec la methode read
    - abstract void **write**(int b)
  - Ecrit un tableau de byte (plus efficace) du tableau b vers le flux
    - void **write**(byte[] b)
    - void **write**(byte[] b, int off, int len)
  - Demande d'écrire ce qu'il y a dans le buffer
    - void **flush**()
  - Ferme le flux
    - void **close**()

Utilisation (Copie de flux)

- Byte par byte (mal)

```
public static void copy(Inputstream in,OutputStream out)
   throws IOException {

   int b;
   while((b=in.read())!=-1)
     out.write(b);
}
```

- Par buffer de bytes

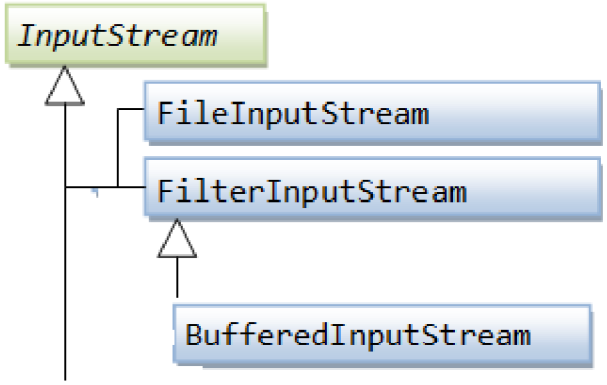
```
public static void copy(Inputstream in,OutputStream out)
   throws IOException {

   byte[] buffer=new byte[8192];
   int size;
   while((size=in.read(buffer))!=-1)
     out.write(buffer,0,size);
}
```

FileInputStream/FileOutputStream

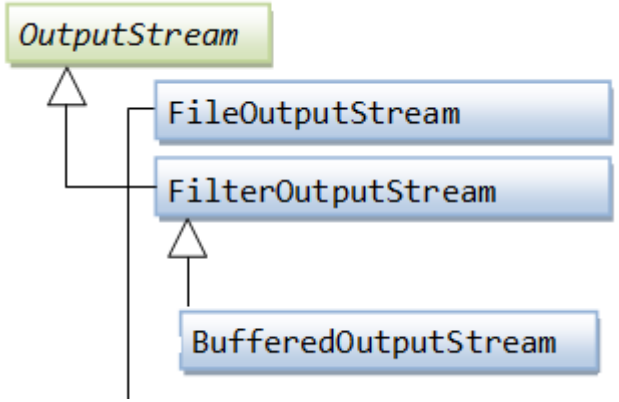
- Pour créer un InputStream sur un fichier
  - **FileInputStream**(String name)
  - **FileInputStream**(File file)

  Renvoie l'exception
  FileNotFoundException qui hérite de **IOException** si le fichier n'existe pas

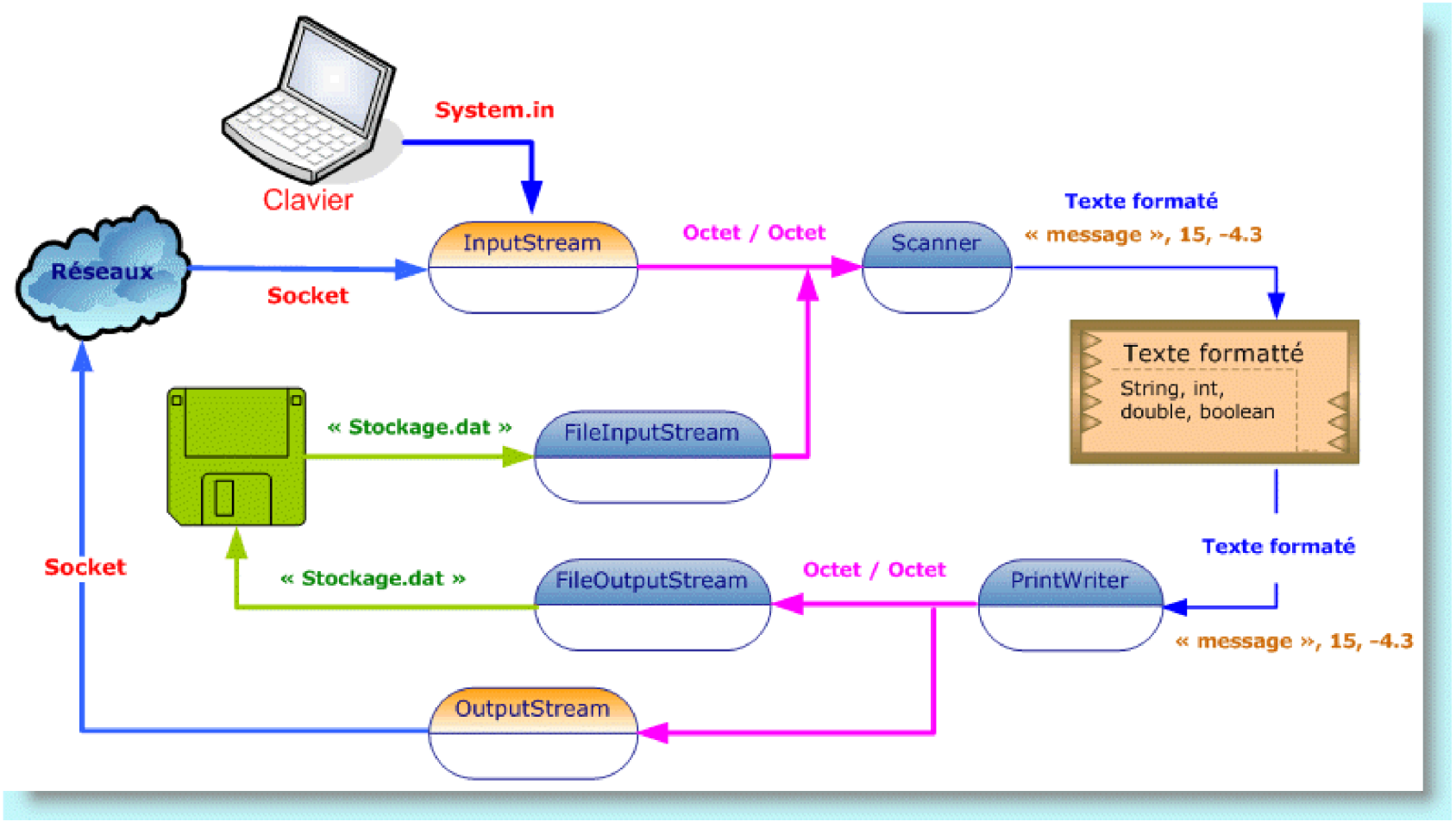


- Pour créer un OutputStream sur un fichier
  - **FileOutputStream**(String name, boolean append)
  - **FileOutputStream**(File file, boolean append)

  append à true si l'on veut écrire à la fin

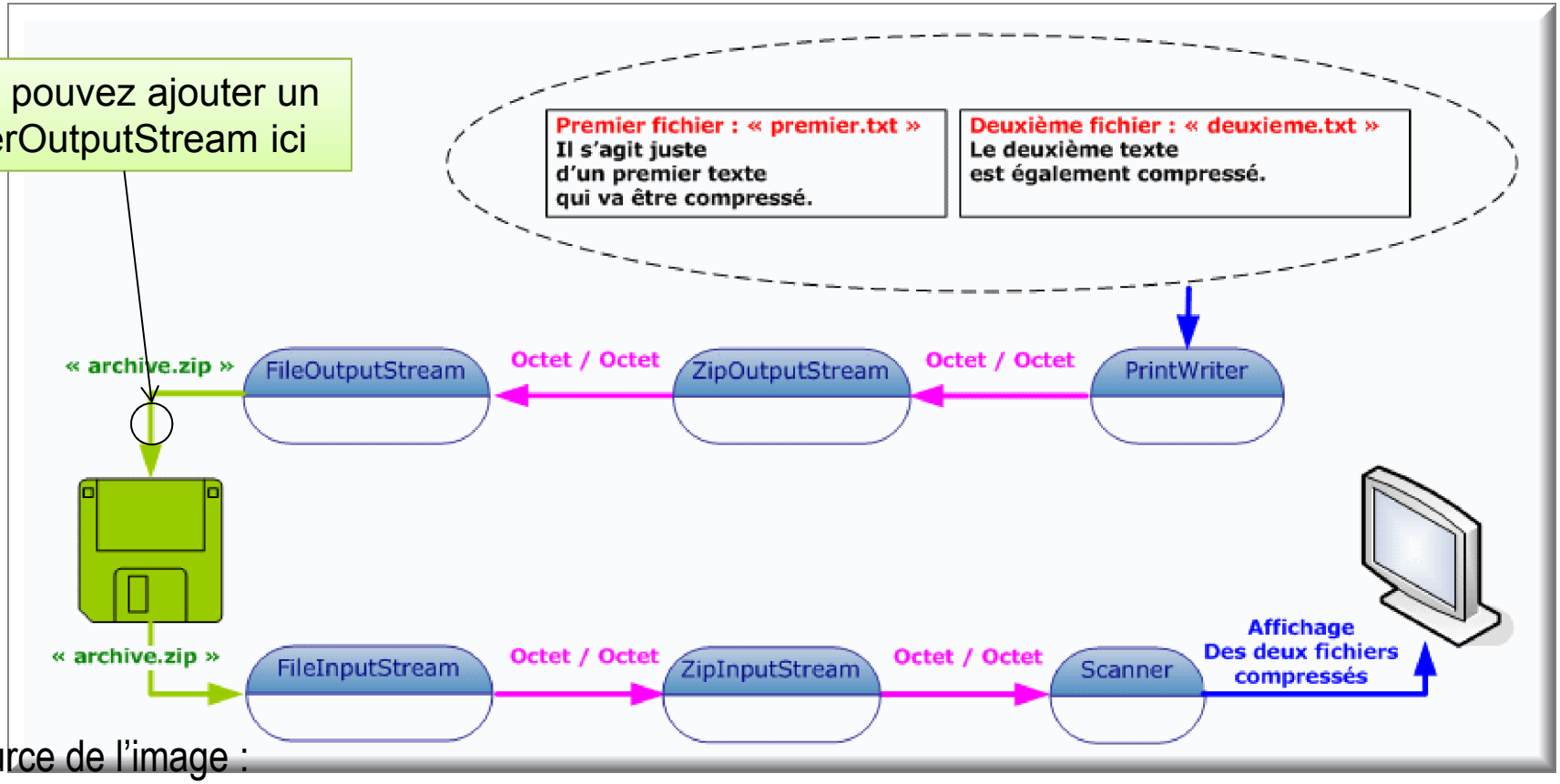


InputStream/OutputStream et Scanner/PrintWriter



Source de l'image :
http://programmation-java.1sur1.com/Java/Tutoriels/ExceptionsFluxFichiers/FluxFichiers.php

Empilement de flux *(Compression et décompression des données)*



Source de l'image :
http://programmation-java.1sur1.com/Java/Tutoriels/ExceptionsFluxFichiers/FluxFichiers.php

Exemple de lecture d'archive .zip

```
import static java.lang.System.*;

public class LireArchive {
public static void main(String[] args) throws FileNotFoundException,
IOException {
ZipInputStream archive = new
                  ZipInputStream(new FileInputStream("archive.zip"));
ZipEntry fichier;
while ((fichier = archive.getNextEntry())!=null) {

 Scanner lecture = new Scanner(archive);
 out.println("Fichier : "+fichier.getName());
 out.println("-----------------------------------------");
 while (lecture.hasNextLine()) {
   out.println(lecture.nextLine()); }
   out.println("-----------------------------------------"); //
    archive.closeEntry();
 }
   archive.close();
 }
}
```

BufferedReader et PrintWriter

- Utiles pour la lecture et l'écriture de fichiers textes.

```
BufferedReader in = new BufferedReader(new FileReader("fichier.txt"));
String str;
while ((str=in.readLine()) != null) {
 // faire quelque chose avec la ligne
}
in.close();

// Pour la lecture au clavier
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));


PrintWriter out = new PrintWriter(new FileWriter("fichier.txt"));
while ((str=in.readLine()) != null) {
        out.println(str);
}
out.close();
```

ObjectInputStream et ObjectOutputStream

- Permettent de lire et écrire des objets sur un stream.
- Un objet doit implémenter l'interface Serializable pour pouvoir être écrit ou lu.
- L'interface Serializable contient les deux méthodes suivantes :
  - private void readObject(java.io.ObjectInputStream stream) throws IOException, ClassNotFoundException;
  - private void writeObject(java.io.ObjectOutputStream stream) throws IOException;

ObjectInputStream et ObjectOutputStream (Exemple)

```
class Etudiant implements Serializable {
        String nom;  String prenom;   double note;
        Etudiant(String n, String p, double n) {
                nom = n;   prenom = p;   note = n;  }
 private void readObject(ObjectInputStream stream) {
        nom = (String) stream.readObject();
        prenom = (String) stream.readObject();
        note = stream.readDouble();
 }
 private void writeObject(ObjectOutputStream stream) {
        stream.writeObject(nom);
        stream.writeObject(prenom);
        stream.writeDouble(note);  } }

Etudiant e = new Etudiant("Ahmed" ,  "Mansor",86);

FileOutputStream fos = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(e);
oos.close();

FileInputStream fis = new FileInputStream("t.tmp");
ObjectInputStream ois = new ObjectInputStream(fis);
Etudiant e2 = (Etudiant) ois.readObject();
ois.close();
```

La généricité

Un type générique est une classe générique ou une interface qui est paramétré sur les types. La classe Boite suivante sera modifié pour illustrer ce concept.

Version non générique → Version générique

```
public class Boite {
   private Object objet;
   public void set(Object objet) {
     this.objet = objet;
   }
   public Object get() {
      return objet;
   }
 public static void main(String [] args){
        Boite b = new Boite();
        b.set(new Bijou());
        Bijou bj = (Bijou) b.get();

}
}
```

```
public class Boite<T>{
   private T objet;
   public void set(T objet) {
     this.objet = objet;
   }
   public T get() {
      return objet;
   }
public static void main(String [] args){
     Boite<Bijou> b = new Boite<Bijou>();
     b.set(new Bijou());
     Bijou bj =  b.get();
}
}
```

Méthode générique

Un exemple montre une méthode générique

```
public static <T> int countEqualsTo(T[] anArray, T elem)  {
          int count = 0;
          for (T e : anArray)
                    if (e.equals(elem))
                              ++count;
          return count;
}
```

Un exemple montre le besoin d'une paramètre de type borné

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)   // compiler error
            ++count;
    return count;
}
```

Paramètre de type borné (solution)

- T doit être de type comparable<T>

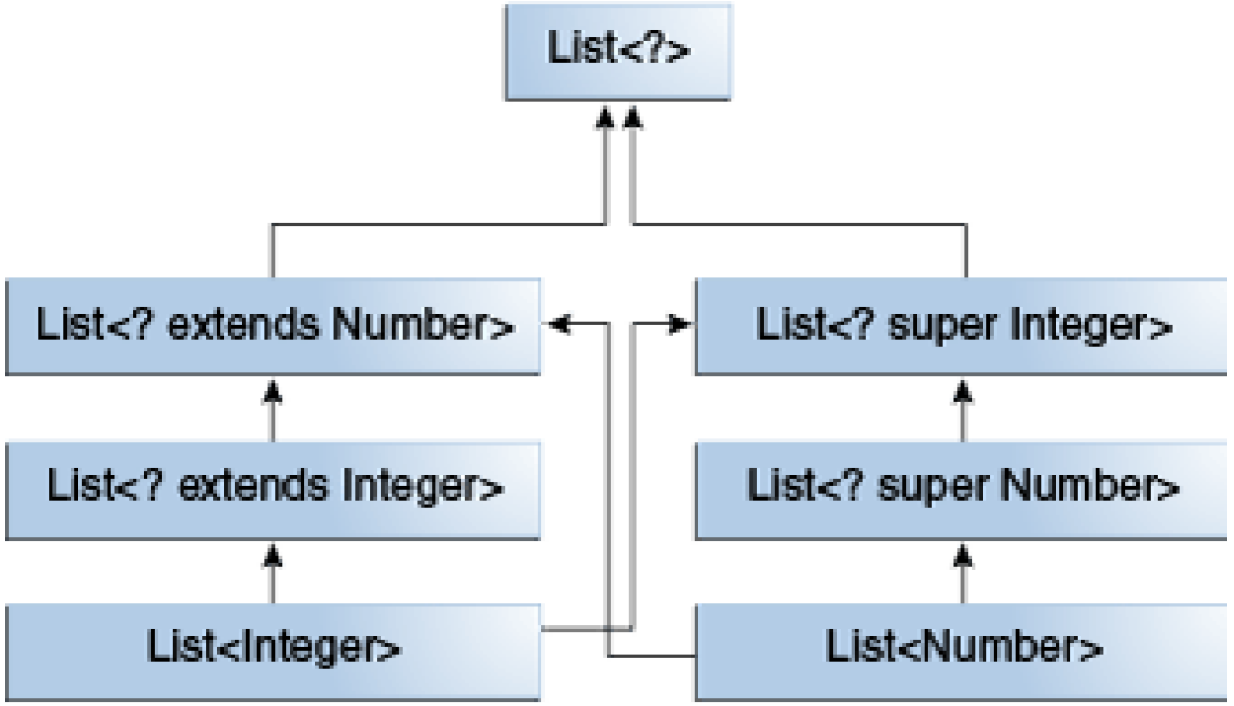
```
public interface Comparable<T> {
     public int compareTo(T o);
}
```

- Code compilable

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Joker (?) / bornes sup et inf

- Exemple: une hiérarchie de plusieurs déclarations de classe de liste générique.



Joker (wildcard) et borne supérieure

- Soit la méthode

```
public static void traiter(List<? extends Foo> list) { /* ... */ }
```

- Le joker de borne supérieure, <? extends Foo>, où Foo est un type quelconque, correspond à Foo ou un sous-type de Foo.

- La méthode traiter ne peut accéder aux éléments de la liste que par le type Foo:

```
public static void traiter(List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...
    }
}
```

Joker (wildcard) et borne inférieure

- Soit la méthode

```
public static void traiter(List<? super Foo> list) { /* ... */ }
```

- Le joker de borne inférieure, <? super Foo>, où Foo est un type quelconque, correspond à Foo ou un super-type de Foo.

- La méthode traiter ne peut accéder aux éléments de la liste que par le type Object:

```
public static void traiter(List<? super Foo> list) {
    for (Object elem : list) {
        // ...
    }
}
```

L'interopérabilité avec les versions enceintes < 1.4

- Type cru (Raw type) est un type d'une classe/interface paramétrée utilisé sans spécifier le paramétre
  - Exmple : List lst = new ArrayList();

- C'est possible mais faire attention! Une ClassCastException peut être levée.

Finalement

- La série de révision N°1
- Les exercices de TPs
  - sont de bonnes moyennes de révision en particulier pour les E/S, les générique et Collections
- N'hésité pas de me contacter si vous avez des questions

red.ezzahir@gmail.com