

# Programmation Avancée et Application

Techniques avancées en Java

---

Jean-Guy Mailly

`jean-guy.mailly@u-paris.fr`

LIPADE - Université de Paris

<http://www.math-info.univ-paris5.fr/~jmailly/>

1. Énumérations
2. Exceptions
3. Types génériques
4. Entrées sorties

# Énumérations

---

- Une énumération est une classe particulière qui a un nombre fini et prédéfini d'instances
- Utilisation basique d'une énumération : l'exemple du jeu de cartes

```
public enum CouleurCarte {  
    COEUR, CARREAU, PIQUE, TREFLE ;  
}
```

- L'énumération est juste une liste de constantes, séparées par des virgules, qui se termine par un point virgule
- Utilisation d'une constante de l'enum :  
CouleurCarte c = CouleurCarte.COEUR ;

# Énumérations et switch

- Les types énumérés peuvent être utilisés dans une instruction **switch-case**

```
CouleurCarte c = CouleurCarte.COEUR;
```

```
switch (c) {  
    case COEUR:  
        System.out.println("C'est_du_coeur."); break;  
    case CARREAU:  
        System.out.println("C'est_du_carreau."); break;  
    case PIQUE:  
        System.out.println("C'est_du_pique."); break;  
    case TREFLE:  
        System.out.println("C'est_du_trefle."); break;  
}
```

## Énumérations plus élaborées : personnages de jeu

```
public enum TypePerso {  
    MAGE(4,7,3), GUERRIER(7,0,7),  
    ELFE(5,2,9), NAIN(6,0,6) ;  
  
    private final int force ;  
    private final int magie ;  
    private final int mouvement ;  
  
    private TypePerso(int force,int magie,int mouvement){  
        this.force = force ;  
        this.magie = magie ;  
        this.mouvement = mouvement ;  
    }  
    public int getForce() { return force; }  
    public int getMagie() { return magie; }  
    public int getMouvement() { return mouvement; }  
}
```

# Méthodes utiles

- `name()` : renvoie un `String` correspondant au nom de la constante

```
public void f(CouleurCarte c){  
    System.out.println("C'est_du_" + c.name() + ".");  
}
```

- `valueOf()` : renvoie la constante correspondant à un `String`

```
CouleurCarte c = CouleurCarte.valueOf("COEUR") ;
```

- `values()` : renvoie un tableau composé des constantes de l'énumération dans l'ordre de leur déclaration
- `ordinal()` : renvoie l'indice d'une constante dans le tableau retourné par la méthode `values()`

```
CouleurCarte[] tab = CouleurCarte.values();  
// [COEUR, CARREAU, PIQUE, TREFLE]  
System.out.println(CouleurCarte.CARREAU.ordinal());  
// Affiche 1
```

# Exceptions

---

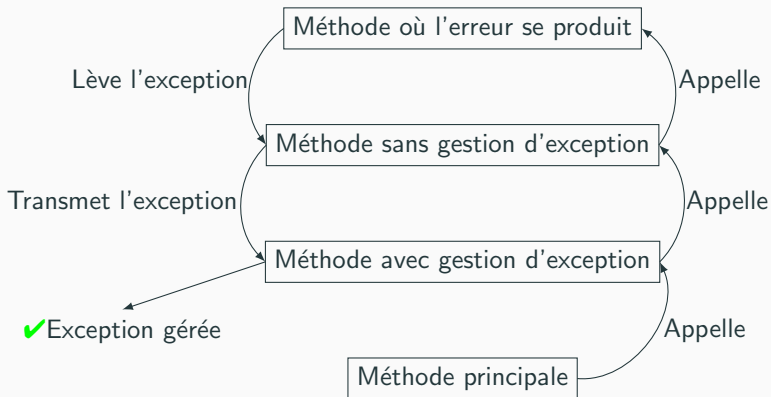


# Qu'est-ce qu'une exception ?

- Une exception dans un programme Java est un comportement inhabituel qui provoque une erreur d'exécution
- Elle peut être gérée pour reprendre la bonne marche du programme, ou éventuellement permettre un arrêt « propre » du programme (par exemple, en fermant les flux ouverts)
- Si une exception n'est pas gérée, elle provoque un arrêt brutal du programme
- Lorsqu'une erreur survient, un objet est créé, qui contient un certain nombre d'informations sur l'erreur : on dit que l'exception est *levée*

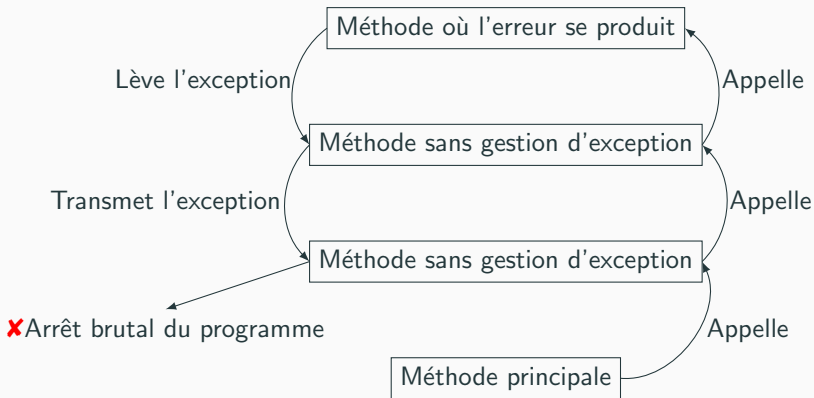
# Gestion d'exception : la pile d'exécution

- L'exception doit être gérée dans une des méthodes de la pile d'exécution qui a mené à l'exception



# Gestion d'exception : la pile d'exécution

- L'exception doit être gérée dans une des méthodes de la pile d'exécution qui a mené à l'exception



# La classe Exception

- Exception est la classe mère de toutes les exceptions. De nombreuses classes filles sont définies
  - IOException : erreurs lors d'entrées/sorties du programme
  - FileNotFoundException : si on essaye d'accéder à un fichier qui n'existe pas (classe fille de IOException)
  - ArrayIndexOutOfBoundsException : si on essaye d'accéder à un indice d'un tableau qui n'existe pas
  - NullPointerException : si on essaye d'utiliser un objet qui vaut **null**
  - NumberFormatException : si on essaye de convertir une String en nombre de manière incorrecte
  - ...
- On peut créer ses propres exceptions en créant une nouvelle classe fille d'Exception ; par convention, les noms des classes d'exceptions se terminent tous par Exception

## Lever une exception (1/4)

```
public static long factorielle(long n) {  
    if (n == 1)  
        return n;  
    else  
        return n * factorielle(n - 1);  
}
```

- Problème : on ne peut pas calculer la factorielle d'un nombre  $\leq 0$
- Solution : lever une exception si l'utilisateur essaye de le faire

## Lever une exception (2/4)

```
public static long factorielle(long n)
                                throws Exception {
    if (n < 1)
        throw new Exception();
    if (n == 1)
        return n;
    else
        return n * factorielle(n - 1);
}
```

- Le mot-clé **throw** suivi d'un objet de type `Exception` permet de lever l'exception
- Le mot-clé **throws** dans la signature indique que la méthode est susceptible de lever une exception

## Lever une exception (3/4)

```
public static long factorielle(long n)
    throws Exception {
    if (n < 1)
        throw new Exception (
            "On ne peut pas calculer la "
            + "factorielle d'un nombre negatif: " + n);
    if (n == 1)
        return n;
    else
        return n * factorielle(n - 1);
}
```

- Il est recommandé de transmettre un message qui décrit précisément l'erreur dans le constructeur de l'exception

## Lever une exception (4/4)

```
public static long factorielle(long n)
    throws MauvaisArgumentException {
    if (n < 1)
        throw new MauvaisArgumentException(
            "On ne peut pas calculer la "
            + "factorielle d'un nombre negatif: " + n);
    if (n == 1)
        return n;
    else
        return n * factorielle(n - 1);
}
```

- Il vaut mieux utiliser le type d'exception le plus spécifique possible pour le problème rencontré, quitte à créer ses propres classes d'exceptions



## Gérer une exception (1/2)

```
public static void calculerDesFactorielles(int n) {  
    for(int i = 1 ; i <= n ; i++) {  
        factorielle(i);  
    }  
}
```

✗ Unhandled exception type MauvaisArgumentException

## Gérer une exception (2/2)

```
public static void calculerDesFactorielles(int n) {  
    for(int i = 1 ; i <= n ; i++) {  
        try {  
            factorielle(i);  
        } catch (MauvaisArgumentException e){  
            // Gestion de l'exception  
        }  
    }  
}
```

- La partie gestion de l'exception peut servir à afficher une information utile à l'utilisateur avant de quitter le programme proprement, ou au contraire à corriger l'erreur

## Rappel : la calculatrice

```
private static void boucle(Scanner sc) {  
    int choix;  
    do {  
        menu();  
        choix = sc.nextInt();  
        switch (choix) {  
            case 0: break;  
            case 1: addition(sc); break;  
            case 2: soustraction(sc); break;  
            default:  
                System.out.println(  
                    "Ce choix n'est pas conforme.");  
        }  
    } while (choix != 0);  
}
```

## La méthode `nextInt()`

- La Javadoc indique la méthode `nextInt` est susceptible de lever plusieurs exceptions, dont :  
`InputMismatchException` : if the next token does not match the Integer regular expression, or is out of range
- On peut donc modifier la méthode `boucle(Scanner sc)` pour s'assurer que l'utilisateur entre bien un nombre entier au clavier

## Gérer les exceptions de la calculatrice

- On doit remplacer la ligne `choix = sc.nextInt ();` :

```
boolean choixEstFait = false ;  
do {  
    try {  
        choix = sc.nextInt ();  
        choixEstFait = true ;  
    } catch (InputMismatchException e) {  
        System.out.println ("Choix_incorrect") ;  
        sc.nextLine ();  
        menu ();  
    }  
} while (! choixEstFait );
```

- Si `sc.nextInt ()` lève une exception, la valeur du booléen n'est pas modifiée, donc on ne sort pas de la boucle

# Gérer les exceptions de la calculatrice



Pour vous entraîner, gérez les exceptions dans les autres méthodes de la calculatrice (addition/soustraction)

# Transmettre une exception

- Il est possible de ne pas gérer une exception dans la méthode où elle se produit : on la laisse passer vers la méthode appelante
- L'exception doit alors être gérée par la méthode appelante, ou alors elle doit être à nouveau explicitement laissée passer

```
public static void calculerDesFactorielles(int n)  
                                throws MauvaisArgumentException {  
    for(int i = 1 ; i <= n ; i++) {  
        factorielle(i);  
    }  
}
```

- Rappel : si on ne fait que laisser passer l'exception sans jamais la gérer, le programme se termine brutalement lorsque l'exception atteint la méthode main

- RuntimeException est la classe mère des exceptions qui peuvent être levées par le comportement normal de la JVM
- Cette classe (et ses filles) ont la particularité d'être des exceptions *unchecked* : il n'est pas obligatoire de les gérer avec un **try-catch** ou de les signaler avec le mot-clé **throws**
- Cela se justifie par le fait que leur levée n'est pas facile à prédire
- Exemples :
  - ArithmeticException
  - ArrayIndexOutOfBoundsException
  - IllegalArgumentException
  - NullPointerException
  - ...



# Multiple catch

- Un même bloc **try** peut être suivi de plusieurs blocs **catch** si plusieurs exceptions peuvent être levées par le même code

```
try {  
    // ...  
} catch ( ... ){  
    // ...  
} catch ( ... ){  
    // ...  
}
```

- Par exemple, une exception peut être levée par le même bloc d'instructions à cause de :
  - l'ouverture d'un fichier
  - le format des données lues dans le fichier
  - on essaye de lire alors que le fichier est terminé
  - ...

# Multiples catch et héritage

- Si un même bloc peut lever des exceptions de type A et B, avec A **extends** B, il faut faire attention à l'ordre des **catch**

```
try {  
    // ...  
} catch (B b) {  
    // Gestion speciale de l'exception B  
} catch (A a) {  
    // Gestion speciale de l'exception A  
}
```

- Si une exception de type A arrive, comme A hérite de B, elle va être gérée par le premier **catch**

# Multiples catch et héritage

- Si un même bloc peut lever des exceptions de type A et B, avec A **extends** B, il faut faire attention à l'ordre des **catch**

```
try {  
    // ...  
} catch (A a) {  
    // Gestion speciale de l'exception A  
} catch (B b) {  
    // Gestion speciale de l'exception B  
}
```

- Les exceptions de type A sont gérées par le **catch** dédié

# Finally

- Le mot-clé **finally** permet de créer un bloc d'instructions qui s'exécute quoi qu'il arrive après le **try-catch**
- Le bloc **finally** permet d'exécuter du code même si une exception imprévue se produit
  - Utile pour le code de « nettoyage », par exemple pour refermer un flux d'entrées/sorties

```
Scanner sc = new Scanner(System.in);  
try {  
    System.out.println(sc.nextInt());  
} catch( ... ) {  
    // ...  
} finally {  
    sc.close();  
}
```

# Types génériques

---

## Le cas des listes

- En Java « à l'ancienne », on faisait `ArrayList l = new ArrayList();`
- Problème : quand on accède à un élément de la liste, on ne peut pas le manipuler sans le caster de la bonne manière. Par exemple, même si on sait que `l` ne contient que des `String`, on ne peut pas faire `String s = l.get(0);` ni

```
Object o = l.get(0) ;  
System.out.println(o.length());
```

- Solution avec un cast :

```
String s = (String)l.get(0) ;  
System.out.println(s.length());
```

# Le cas des listes

- En Java « à l'ancienne », on faisait `ArrayList l = new ArrayList();`
- Problème : quand on accède à un élément de la liste, on ne peut pas le manipuler sans le caster de la bonne manière. Par exemple, même si on sait que `l` ne contient que des `String`, on ne peut pas faire `String s = l.get(0);` ni

```
Object o = l.get(0) ;  
System.out.println(o.length());
```

- Solution avec un cast :

```
String s = (String)l.get(0) ;  
System.out.println(s.length());
```

⚠ Plusieurs problèmes :

- Code peu lisible s'il y a trop de cast
- Risque d'erreur important car on n'est pas sûr à la compilation du type des données manipulées

- Dans la définition des listes, on peut utiliser un paramètre qui décrit le types des éléments de cette liste
- Avantages :
  - Vérification des types dès la compilation : détection et correction d'erreurs plus facile
  - Moins de cast (→ code plus léger)
  - Possibilité d'écrire des algorithmes génériques, c'est-à-dire des algorithmes qui se comportent globalement de la même manière quel que soit le type des données
    - pour trier des nombres dans l'ordre croissant, ou trier des chaînes de caractères dans l'ordre alphabétique, l'algorithme est globalement le même



# Définition d'une classe génériques

- On peut paramétrer la définition d'une classe générique par plusieurs types :

```
public class MaClasse<T1,T2,...,Tn>{  
    // ...  
}
```

- Conventions pour les noms :
  - E : pour des types d'éléments (cf List<E>)
  - K et V : respectivement pour les clés et valeurs (cf Map<K,V>)
  - N : pour des types numériques
  - Pour tout autre type, T (puis S pour le deuxième type, U pour le troisième, V pour le quatrième,...)
- On peut également créer des interfaces génériques (par exemple, List<E> est une interface, tandis que ArrayList<E> est une classe qui implémente List<E>)

## Exemple de classe générique simple

```
public class Couple<T, U> {  
    private T premier ;  
    private U second ;  
  
    public Couple(T p, U s) {  
        premier = p ;  
        second = s ;  
    }  
  
    public T getPremier() {  
        return premier ;  
    }  
  
    public U getSecond() {  
        return second ;  
    }  
}
```

# Instances concrètes de la classe générique

- On peut créer des objets qui appartiennent à différents types de couples en fonction des valeurs données aux paramètres T et U
- Les valeurs peuvent être identiques ou différentes, et peuvent même appartenir à un type paramétré
- Exemples :
  - `Couple<String,String> c1 = new Couple<>("toto","titi");`
  - `Couple<Double,Integer> c2 = new Couple<>(2.3,4);`
  - `Couple<String,Double> c3 = new Couple<>("toto",1.0);`
  - `Couple<Double,Double> c4 = new Couple<>(1.0,3.5);`
  - `Couple<Couple<Double,Double>,Couple<Double,Double>> c5 = new Couple<>(c4,new Couple<>(2.3,5.1));`

## La généricité cachée (1/3)

- On peut créer des classes qui héritent d'une classe générique afin de simplifier certains traitements, et bénéficier des méthodes définies dans la classe générique

```
public class Point extends Couple<Double , Double> {  
    public Point(Double abs , Double ord) {  
        super(abs , ord);  
    }  
  
    public double getAbscisse() {  
        return getPremier();  
    }  
  
    public double getOrdonnee() {  
        return getSecond();  
    }  
}
```

## La généricité cachée (2/3)

```
public class Personne extends Couple<String , String> {  
    public Personne(String prenom , String nom) {  
        super(prenom , nom);  
    }  
  
    public String getPrenom() {  
        return getPremier();  
    }  
  
    public String getNom() {  
        return getSecond();  
    }  
}
```

- On suppose qu'on a défini une classe Etudiant, et on définit une promotion comme étant une liste d'étudiants

```
public class Promotion extends ArrayList<Etudiant> {  
    // ...  
}
```

- On peut utiliser directement toutes les méthodes d'une ArrayList (isEmpty(), add(),...), et ajouter des méthodes propres à une promotion d'étudiants (par exemple calculMoyenneDePromo())

- Il n'est pas nécessaire de préciser les types lors de l'utilisation de **new** si le compilateur peut les déduire du contexte. Exemples :
  - `ArrayList<Integer> l = new ArrayList<>();`
  - `Couple<String,String> c = new Couple<>();`
- Ne pas oublier les chevrons `<>`, sinon cela définit un type brut (*raw type*). Les types bruts sont conservés pour la compatibilité du vieux code (avant l'introduction des types génériques), mais il n'est pas recommandé de les utiliser

# Méthodes génériques

- Une méthode générique est une méthode qui introduit un (ou des) type(s) paramétré(s).
- Une méthode générique peut être définie dans une classe générique, ou dans une classe non générique
- Les paramètres sont introduits entre chevrons avant le type de retour de la méthode

```
public class Util {  
    public static <T, U> boolean compareCouples(  
        Couple<T, U> c1, Couple<T, U> c2) {  
        return c1.getPremier().equals(c2.getPremier())  
            && c1.getSecond().equals(c2.getSecond());  
    }  
}
```



- Syntaxe complète :

```
Couple<String , Double> c1 = new Couple<>("toto" , 2.3);  
Couple<String , Double> c2 = new Couple<>("titi" , 2.2);  
boolean b = Util.<String , Double>compareCouples(c1 , c2 );
```

- Syntaxe complète :

```
Couple<String , Double> c1 = new Couple<>("toto" , 2.3);  
Couple<String , Double> c2 = new Couple<>("titi" , 2.2);  
boolean b = Util.<String , Double>compareCouples(c1 , c2 );
```

- Syntaxe allégée grâce à l'inférence de types du compilateur :

```
Couple<String , Double> c1 = new Couple<>("toto" , 2.3);  
Couple<String , Double> c2 = new Couple<>("titi" , 2.2);  
boolean b = Util.compareCouples(c1 , c2 );
```

- Supposons qu'on veut écrire un algorithme qui dépend de la méthode `f()` qui est définie dans la classe (ou l'interface) `A`
- Différentes classes héritent de la classe `A` (ou implémentent) l'interface `A`, et on ne sait pas à l'avance quels types seront utilisés lors de l'exécution
- On peut indiquer dans la signature d'une méthode qu'on a besoin d'utiliser un type paramétré qui est un sous-type de `A`
- On va montrer sur un exemple qu'on ne peut pas toujours utiliser directement `A` quand on a besoin de ses sous-types

## Types bornés : intuition

```
public void algo(List<A> l){  
    for(A a : l){  
        a.f() ;  
    }  
}
```

- Soit A1 une classe qui hérite de A, et l une liste d'éléments de type A1. L'instruction algo(l) ; provoque l'erreur suivante :

The method algo(List<A>) in the type Util is not applicable for the arguments (List<A1>)

- **Remarque** : Même quand A1 est un sous-type de A, MaClasse<A1> et MaClasse<A> ne sont pas parentes

## Types bornés : borne supérieure

- On peut indiquer que les éléments de `I` doivent appartenir à un sous-type de `A`
- On utilise le mot-clé **extends**, que `A` soit une classe ou une interface

```
public <T extends A> void algo(List<T> I){  
    for(T a : I){  
        a.f();  
    }  
}
```

- On peut maintenant appeler `algo(I)` ; avec `I` de type `List<A>`, `List<A1>`, ou n'importe quel autre type `List<XXXX>`, avec `XXXX` un sous-type de `A`

# Généricité et héritage

- *Même quand A1 est un sous-type de A, MaClasse<A1> et MaClasse<A> ne sont pas parentes : explication sur un exemple concret*

```
List<Integer> l1 = new ArrayList<Integer>() ;  
List<Number> l2 = l1 ;
```

✗Erreur :

Type mismatch: cannot convert from List<Integer> to List<Number>

- Comme l2 est de type List<Number>, on devrait pouvoir ajouter dedans n'importe quel objet de type Number
- Problème : concrètement, l2 est une référence vers un objet de type ArrayList<Integer>, on ne peut donc pas ajouter n'importe quel Number dedans

## Types bornés : bornes multiples

- On peut spécifier plusieurs bornes supérieures :  
`<T extends A & B & C>`
- Les bornes supérieures peuvent être (éventuellement) une classe-mère, et des interfaces
- Si une des bornes est une classe-mère, elle doit être donnée en première position de la liste :

```
public class A { ... }  
public interface B { ... }  
public interface C { ... }
```

```
public <T extends B & C & A> f(){ ... }
```

✗ Ne fonctionne pas car la classe A n'est pas donnée en premier

## Types bornés : bornes multiples

- On peut spécifier plusieurs bornes supérieures :  
`<T extends A & B & C>`
- Les bornes supérieures peuvent être (éventuellement) une classe-mère, et des interfaces
- Si une des bornes est une classe-mère, elle doit être donnée en première position de la liste :

```
public class A { ... }  
public interface B { ... }  
public interface C { ... }
```

```
public <T extends A & B & C> f(){ ... }
```

✓ Ok : la classe A est donnée en premier



- Le point d'interrogation est un joker (en anglais, *wildcard*) pour représenter un type inconnu
- On peut définir une méthode générique qui fonctionne avec n'importe quel sous-type d'un type A, sans avoir besoin de déclarer un type en paramètre :

```
public void f(List<? extends Number> l){  
    for(Number n : l){  
        // ...  
    }  
}
```

- Cette méthode fonctionne avec différents types de paramètres  
List<Number>, List<Integer>, List<Double>,...

## Jokers : borne inférieure

- On peut indiquer avec un joker qu'on a besoin d'objets qui appartiennent à un type parent d'un d'autre type
- Exemple : pour tester si tous les éléments d'une liste d'Integer apparaissent dans une certaine collection, on peut limiter les objets de la collection à un type parent d'Integer (Number or Object) :

```
public static boolean containsAll(List<Integer> l,  
                                   Collection<? super Integer> c) {  
    for (Integer i : l) {  
        if (!c.contains(i))  
            return false;  
    }  
    return true;  
}
```

# Entrées sorties

---

- Rappel : on a vu précédemment les entrées-sorties de base (System.out et System.in + Scanner)
- Nous allons voir ici quelques approches plus élaborées, notamment pour la manipulation de fichiers
- En plus des classes et méthodes que nous verrons dans ce cours, d'autres outils sont fournis par le package java.io

- Classe abstraite, mère de toutes les classes qui correspondent à des flux d'entrée d'octets
- La méthode abstrait `read()` doit être définie par les classes filles concrètes ; elle retourne un **int** qui correspond au prochain octet à lire dans le flux d'entrée
- Entrée de bas niveau : on récupère directement les octets sans tenir compte du genre de données manipulé
- On verra les classes filles `FileInputStream` et `ObjectInputStream`

# OutputStream : la base de l'écriture de données

- Contrepartie d'InputStream : classe abstraite, mère de toutes les classes qui correspondent à des flux de sortie d'octets
- La méthode abstraite `write(int b)` doit être définie par les classes filles concrètes ; elle écrit un octet donnée sous forme d'**int** dans le flux de sortie
- Sortie de bas niveau : on récupère directement les octets sans tenir compte du genre de données manipulé
- On verra notamment les classes filles `FileOutputStream` et `ObjectOutputStream`

## PrintStream : ajout de fonctionnalités à un OutputStream

- Les différentes version de la méthode `print (...)` permettent d'écrire des données de différents types dans flux de sortie

```
OutputStream os = ... ;  
// n'importe quel type d'OutputStream  
PrintStream ps = new PrintStream(os) ;  
ps.print('c') ;  
ps.print(3.2) ;  
ps.print("toto") ;  
ps.print(new MaClasse()) ;
```

- Les versions de `println (...)` permettent la même chose, avec l'ajout d'un retour à la ligne après l'écriture des données passées en paramètre
- L'objet `System.out` est une instance de `PrintStream`

# Manipulation de fichiers : la classe File

- File permet de manipuler des fichiers et des répertoires grâce au chemin qui les identifie
- L'attribut separator permet de construire un chemin en utilisant le séparateur adapté au système d'exploitation (/ sous Unix, \ sous Windows)
- Exemple : on veut ouvrir le fichier toto.txt, situé dans le répertoire rep, lui-même situé dans le répertoire repParent :

```
File f = new File("repParent" + File.separator  
                  + "rep" + File.separator + "toto.txt");
```



## Quelques méthodes de la classe File

- `canRead/canWrite/canExecute` : testent si le programme a les droits de lecture/écriture/exécution sur le fichier
- `createNewFile()/mkdir()` : crée un fichier/répertoire s'il n'existe pas déjà
- `delete()` : supprime un fichier ou répertoire
- `exists()` : teste si le fichier existe
- `getParent()` : retourne le chemin vers le parent du fichier sous forme de `String`
- `isDirectory()/isFile()` : testent si le fichier est un répertoire ou un fichier ordinaire
- ...

De nombreuses autres méthodes sont décrites dans la Javadoc :  
`java.io.File`

- `FileInputStream` et `FileOutputStream` : lecture/écriture de données brutes
- Instances construites avec un `File` en paramètre, ou un `String`
- La lecture et l'écriture se font avec les méthodes de bas niveau héritées des classes `InputStream` et `OutputStream`
- Utile si on doit manipuler directement des données non textuelles (image, son, fichier exécutable,...)

# Sérialisation

- La sérialisation est l'écriture de données représentées sous forme d'objets dans un flux de sortie :
  - dans un fichier pour la persistance des données
  - dans une socket réseau pour la communication entre plusieurs machines
  - ...
- Utilisation de la classe `ObjectOutputStream`

```
FileOutputStream fos = new FileOutputStream(  
    "toto.txt");  
ObjectOutputStream oos = new ObjectOutputStream(  
    fos);  
oos.writeObject(new MaClasse());
```

- Si on souhaite sérialiser une instance d'une classe A, tous les attributs de A doivent être eux-mêmes sérialisables, sauf s'ils sont statiques ou identifiés par le mot-clé **transient**

- Le processus inverse (→ décodage des données depuis un flux pour reformer un objet Java) est appelé désérialisation
- Utilisation de la classe `ObjectInputStream`
- La désérialisation avec `readObject()` de la classe `ObjectInputStream` retourne un `Object`, il faut donc faire un transtypage vers le type souhaité

```
FileInputStream fis = new FileInputStream(  
    "toto.txt");  
ObjectInputStream ois = new ObjectInputStream(fis);  
MaClasse mc = (MaClasse) ois.readObject();
```

- Si on désérialise un objet dont un attribut est **transient**, cet attribut est initialisé à **null**

# L'interface Serializable

- On ne peut pas sérialiser un objet si la classe à laquelle il appartient n'implémente pas l'interface `Serializable`
- Cette interface n'a pas d'attributs ni de méthodes, elle sert uniquement à indiquer qu'un objet peut être sérialisé/désérialisé avec les méthodes `writeObject` et `readObject` des classes `ObjectOutputStream` et `ObjectInputStream`

- Classes abstraites, respectivement mères des classes dédiées à la lecture et à l'écriture de flux de caractères
- Pour des données textuelles, il n'y a pas besoin de gérer soi-même l'encodage des caractères en octets
- `read(char[] buf)` de la classe `Reader` lit des caractères jusqu'à la fin du flux d'entrée (ou l'arrivée d'une erreur), et les stocke dans le tableau `buf` passé en paramètre
- La classe `Writer` fournit (entre autres) les méthodes `write(char[] buf)` et `write(String s)` pour écrire des caractères (ou une chaîne de caractères) dans un flux

## Faire le lien entre IO de bas niveau et IO de haut niveau

- `InputStreamReader` est une classe fille de `Reader` qui permet de passer d'un flux d'octets ( $\rightarrow$  `InputStream`) à un flux de caractères
- La conversion est faite grâce à la connaissance de l'encodage (*charset*) utilisé, qui peut être donné en paramètre (ou par défaut, celui du système)
- De manière symétrique, `OutputStreamWriter` est une classe fille de `Writer` qui permet de convertir des caractères en octets pour les transmettre à un `OutputStream`

- `FileReader` et `FileWriter` sont des classes filles (respectivement) d'`InputStreamReader` et `OutputStreamWriter`, qui sont dédiées à la lecture et l'écriture dans des fichiers
- Contrairement aux `FileInputStream` et `FileOutputStream`, ces classes sont adaptées à la lecture/écriture de données textuelles (→ caractères)



## Lecture bufferisée (1/2)

- `BufferedReader` est une classe fille de `Reader` qui propose une lecture efficace de caractères et de lignes dans un flux d'entrée
- un `BufferedReader` peut uniquement lire dans un autre `Reader`, par exemple :
  - Dans un `InputStreamReader` pour faire le lien avec un `InputStream` quelconque

```
InputStream is = ... ;  
InputStreamReader isr = new InputStreamReader(is);  
BufferedReader bReader = new BufferedReader(isr);
```

- Dans un `FileReader`

```
FileReader fReader = new FileReader("toto.txt");  
BufferedReader bReader = new BufferedReader(fReader);
```

## Lecture bufferisée (2/2)

- `readLine()` est une méthode de `BufferedReader` qui lit une ligne à partir du flux d'entrée et la retourne sous forme de `String`
- `readLine()` retourne **null** lorsque la fin du fichier est atteinte
- Un exemple de procédure standard pour lire les lignes d'un fichier est donc :

```
FileReader fReader = new FileReader("toto.txt");  
BufferedReader bReader=new BufferedReader(fReader);  
String ligne = null;
```

```
while((ligne = bReader.readLine()) != null){  
    // faire quelque chose avec la String obtenue  
    ...  
}
```

# Écriture bufferisée

- De manière symétrique, `BufferedWriter` est une classe fille de `Writer` qui propose une écriture efficace de caractères dans un flux de sortie
- un `BufferedWriter` peut uniquement écrire dans un autre `Writer`, et rend les écritures plus efficaces :

```
BufferedWriter bW = new BufferedWriter(  
    new FileWriter("toto.txt")) ;  
PrintWriter pW = new PrintWriter(bW) ;
```

- La classe `PrintWriter` fournit des méthodes d'écriture similaires à celles de `PrintStream` (`print (...)` / `println (...)`)
- Sans le `BufferedWriter`, l'appel à `pw.println (...)` nécessiterait, pour chaque caractère à écrire, sa conversion en octet et son écriture sur le fichier → nombreux accès au disque dur
- Avec le `BufferedWriter`, les caractères sont convertis en octets et écrits sur le fichier par « paquets » → moins d'accès au disque dur