

Licence Professionnelle

Ingénierie des Systèmes Informatiques et Logiciels

(ISIL)

PROGRAMMATION ORIENTÉE OBJET

(JAVA)

Pr. Said BENKIRANE
2016/2017

Partie 3: Les Exceptions, Entrées/Sorties, Collections, Enumérations et threads.

- Les Exceptions
- Les Entrées/Sorties
- Les Collections
- Les Enumérations
- Les Threads



Les Exceptions

Exceptions

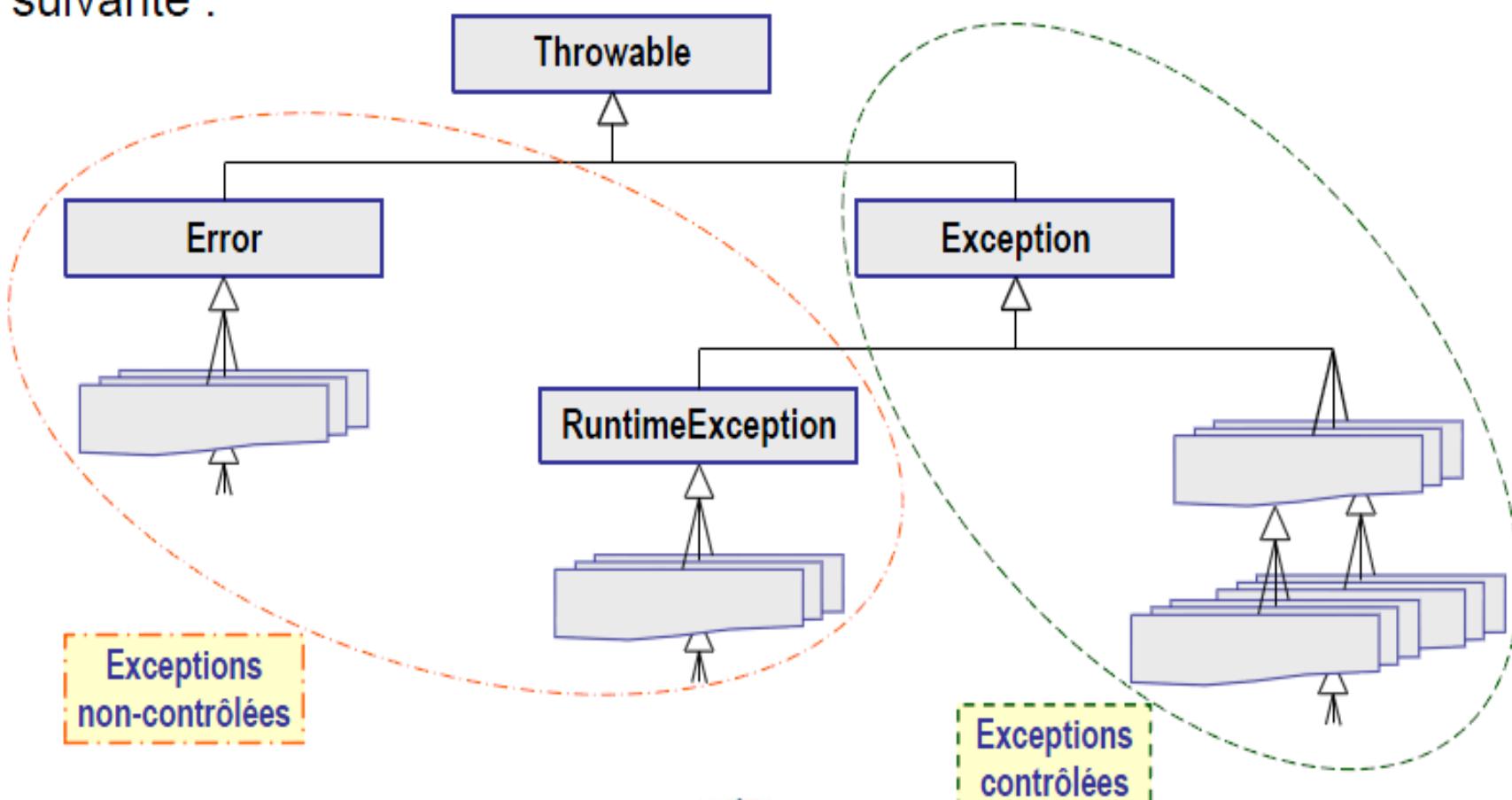
- Les **exceptions** représentent des **événements** qui peuvent survenir durant l'exécution d'un programme et qui **rompent le flux normal** des instructions.
- Les **exceptions** sont principalement utilisées pour représenter des **erreurs** de différents types : des erreurs matérielles (crash du disque, ...) des erreurs de programmation (indice d'un tableau hors limites, ...) des erreurs liées à l'environnement d'exécution (mémoire insuffisante, ...), des erreurs spécifiques à une librairie (matrice singulière) ou à une application (numéro d'article non-défini), etc.
- Les **exceptions** peuvent également représenter des événements qui en sont pas à proprement parler des erreurs, mais qui correspondent à des **situations exceptionnelles** (prévues) qui doivent être traitées de manières différentes du flux normal des opérations (par exemple la fin d'un fichier, un mot de passe incorrect, des ressources momentanément non-disponibles, ...).

Avantages des exceptions

- Augmentent la **lisibilité** du code en séparant les instructions qui traitent le cas normal des instructions nécessaires au traitement des erreurs ou des événements exceptionnels.
- Permettent (voire imposent) la **déclaration explicite** des exceptions qu'une méthode peut lever (fait partie de la signature).
- Forcent le programmeur à **prendre en compte** (traiter ou propager) les cas exceptionnels (erreurs ou autres événements) qui sont déclarés dans les méthodes qu'il invoque.
- Permettent de créer une **hiérarchie** d'événements et de les traiter avec une granularité différente selon les situations.
- Offrent un mécanisme de **propagation automatique** ce qui permet au programmeur de choisir à quel niveau il souhaite traiter l'exception (celui auquel il est à même de prendre les mesures adéquates).

Hiérarchie de classes

- Dans le langage Java, toutes les exceptions sont représentées par des objets qui spécialisent la classe de base **Throwable**.
- Trois sous-classes de **Throwable** sont pré-définies : **Error**, **Exception** et **RuntimeException** selon l'arborescence suivante :



Exceptions contrôlées / non-contrôlées

- Les **exceptions contrôlées (checked)** [du type *Exception*] doivent être soit traitées (dans une clause `catch`), soit propagées pour être traitées à un niveau supérieur (annoncée avec `throws` dans l'en-tête).
- Elles ne peuvent pas être ignorées (le compilateur génère une erreur dans ce cas).
- Pour les **exceptions non-contrôlées (unchecked)** [du type *RuntimeException* ou *Error*], le programmeur a le choix de les traiter ou de les ignorer (sans erreur à la compilation).
- Si une telle exception survient et qu'elle n'est traitée nulle part, elle sera alors propagée et "remontera" jusqu'à la méthode `main()` interrompant ainsi brutalement l'application.

La classe Error

- La classe **Error** représente généralement des erreurs sévères qui surviennent dans la machine virtuelle.
- Le langage n'impose pas que les programmeurs traitent ce type d'exceptions car elles ne devraient pas survenir dans un environnement sain (machine virtuelle correctement installée).
- On ne traite généralement pas ce type d'exceptions dans une application standard sauf éventuellement au niveau le plus haut pour informer l'utilisateur (si c'est encore possible) et éviter le crash brutal de l'application.
- D'autre part, les applications ne devraient pas générer (dans des instructions **throw**) ce genre d'exceptions.
- Exemples : **OutOfMemoryError**, **StackOverflowError**,
AssertionError (Erreur de conception)

La classe Exception

- La classe **Exception** représente la classe de base de la plupart des exceptions qui sont générées (**throw**) et traitées (**catch**) dans les applications (exceptions contrôlées).
- Un grand nombre de sous-classes de **Exception** sont pré-définies dans les classes de la plate-forme Java
- Exemples : **PrintException**, **IOException**
- Les exceptions spécifiques à une librairie ou à une application sont généralement représentées par des sous-classes de **Exception**.
- La classe **RuntimeException** représente une sous-classe particulière de **Exception** (voir page suivante).
- Le langage impose que les exceptions de type **Exception** (qui ne sont pas une sous-classe de **RuntimeException**) soient traitées par les applications (en plaçant les instructions critiques dans un bloc **try / catch**, ou en déclarant l'exception dans la signature de la méthode à l'aide du mot-clé **throws** déléguant ainsi le traitement au niveau supérieur).

La classe `RuntimeException`

- La classe `RuntimeException` représente une sous-classe particulière (exceptions non-contrôlées) de la classe `Exception`.
- Elle représente des erreurs qui sont détectées par la machine virtuelle durant l'exécution de l'application.
Par exemple, l'exception `NullPointerException` indique que la référence à un objet n'est pas définie (`null`).
Cette exception peut potentiellement survenir lors de l'exécution de n'importe quelle instruction qui manipule un objet.
- De ce fait, le langage n'impose pas que les exceptions de type `RuntimeException` (et de ses sous-classes) soient traitées par les applications (ce qui serait extrêmement lourd et contraignant).
- Si nécessaire, les applications peuvent cependant traiter ce type d'exceptions (dans un bloc `try / catch`) et prendre ainsi les mesures adéquates.
- Exemples : `ArithmetricException`,
`ArrayIndexOutOfBoundsException`

Gestion des exceptions

- En Java, les exceptions sont représentées par des objets de type **Throwable**. Il existe plusieurs familles d'exceptions représentées par différentes sous-classes de **Throwable** (par ex. **Exception**).
- Différentes instructions permettent de gérer les exceptions.
- L'instruction **throw** sert à **générer une exception** (on dit également **lever une exception**, **lancer une exception**, ...).
- L'instruction **try / catch / finally** constitue le cadre dans lequel les exceptions peuvent être détectées (capturées) et traitées.
- Le mot-clé **throws** (à ne pas confondre avec **throw**) est utilisé dans la déclaration de méthode (signature) pour **annoncer la liste des exceptions** que la méthode peut générer. L'utilisateur de la méthode est ainsi informé des exceptions qui peuvent survenir lors de son invocation et peut prendre les mesures nécessaires pour gérer ces événements exceptionnels (c'est-à-dire les **traiter** ou les **propager**).

Générer (lever) une exception [1]

- Les exceptions peuvent être générées soit :
 - par le **système**, lors de l'exécution de certaines instructions
 - par l'exécution de l'**instruction throw** (dans les classes pré-définies de la plate-forme Java [bibliothèques] ou dans le code que l'on écrit soi-même)
- Parmi les instructions qui génèrent des exceptions, on peut citer :
 - La division entière par zéro qui génère
ArithmeticException
 - L'indexation d'un tableau hors de ses limites qui génère
ArrayIndexOutOfBoundsException
 - L'utilisation d'un tableau ou objet dont la référence vaut **null** qui génère
NullPointerException
 - La création d'un tableau avec une taille négative qui génère
NegativeArraySizeException
 - La conversion d'un objet dans un type non compatible qui génère
ClassCastException
 - ...

Générer (lever) une exception [2]

- Pour générer explicitement une exception, on utilise l'instruction **throw** :
throw exception_object ;
- L'expression qui suit l'instruction **throw** doit être un objet qui représente une exception (un objet de type **Throwable**).
- Lors de la création de l'objet exception, on peut généralement lui associer un **message (String)** qui décrit l'événement.

```
public static long factorial(int x) throws Exception {  
    long result = 1;  
    if (x<0)  
        throw new Exception("x doit être >= 0");  
    while (x > 1) {  
        result *= x;  
        x--;  
    }  
    return result;  
}
```

Traiter une exception [1]

- Les instructions susceptibles de lever des exceptions peuvent être insérées dans un bloc **try / catch** qui se présente de la manière suivante :

```
try {  
    // Bloc contenant des instructions pouvant générer des exceptions.  
}  
catch (ExceptionType1 e1) {  
    // Bloc contenant les instructions qui traitent (capturent) les exceptions  
    // du type ExceptionType1 (ou d'une de ses sous-classes)  
    // On peut référencer l'objet exception à l'aide de la variable e1.  
}  
catch (ExceptionType2 e2) {  
    // Bloc contenant les instructions qui traitent (capturent) les exceptions  
    // du type ExceptionType2 (ou d'une de ses sous-classes).  
    // On peut référencer l'objet exception à l'aide de la variable e2.  
}  
catch (...) {  
    ...  
    // Et ainsi de suite...  
}
```

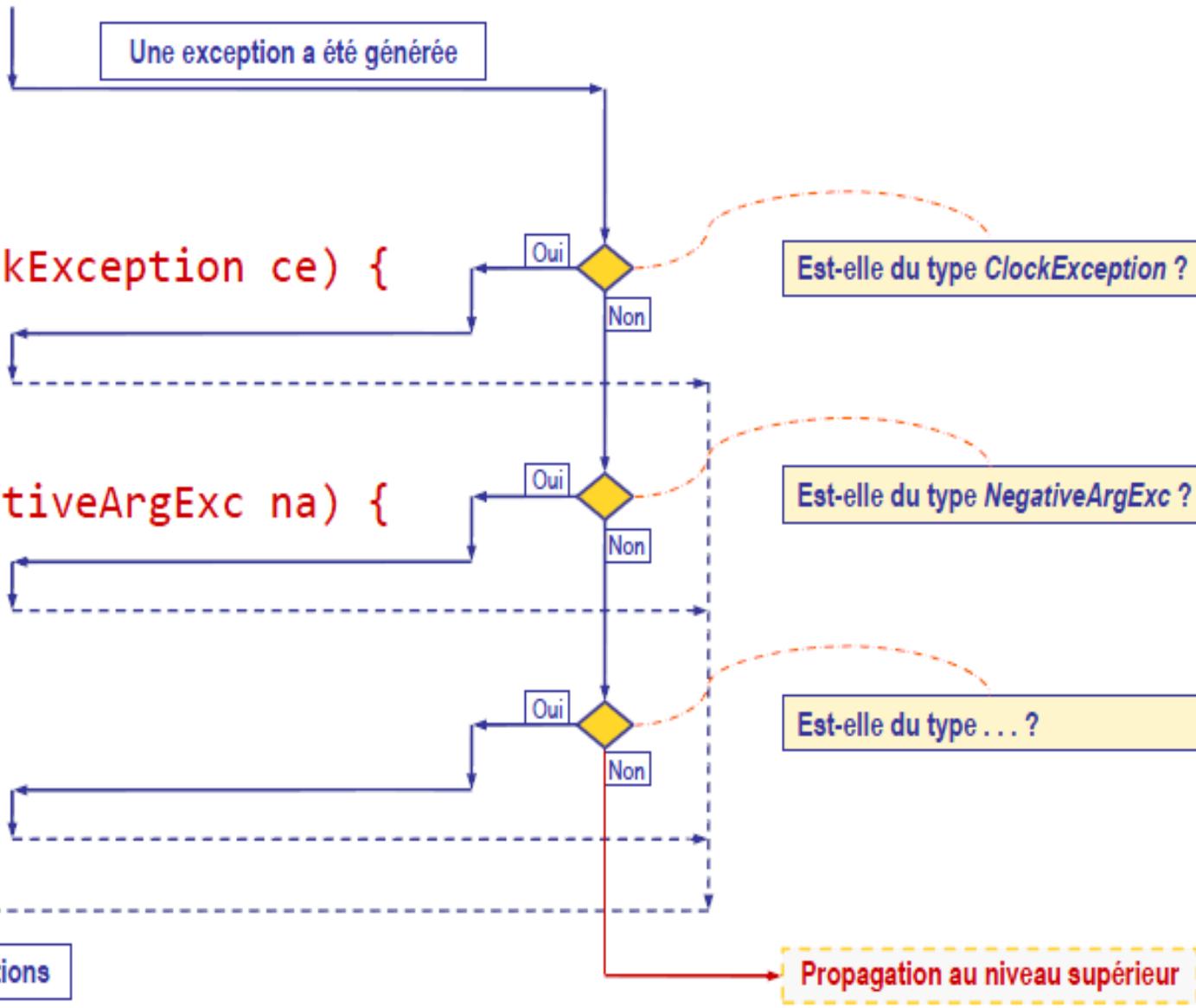
Traiter une exception [2]

- Si une des instructions contenues dans le bloc **try**, lève une exception, le contrôle est passé au premier bloc **catch** dont le type d'exception correspond à l'exception qui a été levée (même classe ou classe parente de l'exception levée).
- Après l'exécution de la dernière instruction du bloc **catch** considéré, le contrôle est passé à l'instruction qui suit le dernier bloc **catch** (ou à la clause **finally** s'il y en a une).
- Si aucun bloc **catch** ne correspond au type d'exception qui a été levée, **l'exception est propagée** au niveau supérieur c'est-à-dire que le contrôle est transféré au traitement d'exception de la méthode invoquante ou du bloc englobant. Si aucun traitement n'existe au niveau supérieur pour ce type d'exception, la propagation se poursuit jusqu'à trouver un bloc **catch** traitant cette exception. Si ce n'est pas le cas, le programme se termine avec un message d'erreur sur la console de sortie (*Stack Trace*).

Traiter une exception [3]

```
try {  
    instr1;  
    instr2;  
    instr3;  
}  
catch (ClockException ce) {  
    instr4;  
    instr5;  
}  
catch (NegativeArgExc na) {  
    instr6;  
    instr7;  
}  
catch (...) {  
    ...  
}
```

Une exception a été générée



Traiter une exception [4]

- Les divers types d'exception peuvent appartenir à **différentes familles** (voir hiérarchie de classes, en fin de chapitre).
- Une exception spécialisée (par ex. `FileNotFoundException`) peut faire partie d'une famille plus vaste (`IOException`) qui elle-même fait partie d'une famille plus générale (`Exception`) etc.
- Si plusieurs clause **catch** sont compatibles avec le type d'exception qui a été levée (font partie de la même famille), **c'est la première qui capturera l'exception** et se chargera du traitement.

```
try {  
    ...  
}  
catch (FileNotFoundException notFound) {  
    ...  
}  
catch (IOException ioErr) {  
    ...  
}  
catch (Exception genErr) {  
    ...  
}
```

L'ordre des clauses
catch est important !

Traiter une exception [5]

- Il est possible de traiter plusieurs types d'exceptions dans une seule clause `catch` en utilisant le symbole '`|`' comme séparateur.
- Les exceptions mentionnées dans une telle liste doivent être "indépendantes" sur le plan de leur relation hiérarchique. Une des exceptions ne peut pas être un ancêtre (*super-classe*) d'une autre (dans ce cas, mentionner l'exception ancêtre suffit pour traiter toutes celles de sa "famille").

```
try {  
    . . .  
}  
catch (UnknownUser | IncorrectPassword loginError) {  
    . . .  
}  
catch (Exception otherError) {  
    . . .  
}
```

Traiter une exception [6]

- Lorsqu'un problème a été détecté (une exception a été générée) et que l'on est en mesure de le traiter (au moins partiellement), il y a **différentes mesures** que l'on peut envisager, selon les cas, pour régler la situation.
- Dans un traitement d'exception (bloc `catch`) on peut par exemple :
 - Régler le problème et recommencer le traitement (nécessite généralement un boucle). Idéal mais pas toujours possible.
 - Faire quelque chose d'autre à la place (algorithme de substitution).
 - Sortir de l'application (`System.exit()`) après affichage d'un message (et/ou de la *Stack-Trace*), écriture dans un fichier *log*, etc.
 - Re-générer l'exception (après avoir effectué certaines opérations).
 - Générer une nouvelle exception (après avoir éventuellement effectué certaines opérations).
 - Retourner une valeur spéciale ou valeur par défaut (pour une fonction).
 - Terminer la méthode (si elle n'a pas de valeur de retour).
 - Ne rien faire (ou afficher un message) et continuer (c'est rarement une bonne solution).

Propagation des exceptions [1]

```
public class PropEx {  
    public static void main(String[] args) {  
        System.out.println("main starts");  
        b(5);  
        b(0);  
        System.out.println("main ends");  
    }  
    //-----  
    //  b  
    //-----  
    public static void b(int i) {  
        System.out.println("b starts");  
        try {  
            System.out.println("b step 1");  
            c(i);  
            System.out.println("b step 2");  
        }  
        catch (Exception e) {  
            System.out.println("b catches " + e);  
        }  
        System.out.println("b ends");  
    }  
}
```

Propagation des exceptions [2]

// ...Suite

```
//-----
//  c
//-----
public static void c(int i) throws Exception {
    System.out.println("c starts");
    d(i);
    System.out.println("c ends");
}

//-----
//  d
//-----
public static void d(int i) throws Exception {
    System.out.println("d starts");
    int a=10/i; // Cette instruction peut générer une exception
    System.out.println("d ends");
}
```

Propagation des exceptions [3]

- Dans le programme PropEx, lors de la deuxième invocation de la méthode `b()`, une exception sera levée dans la méthode `d()` (division par zéro).
- Le déroulement de l'application sera alors altéré et **l'exception sera propagée** jusqu'à la méthode `b()` qui dispose d'un bloc `catch` pour traiter cette exception.

b(5) 1^{ère} invocation	b(0) 2^{ème} invocation
main starts	...
b starts	b starts
b step 1	b step 1
c starts	c starts
d starts	d starts
d ends	b catches ArithmeticException: / by zero
c ends	b ends
b step 2	main ends
b ends	
...	

Interprétation de la *Stack-Trace* [1]

- Si, dans l'exemple précédent (**PropEx**), on supprime tout traitement d'exception, l'exception provoquée par la division par zéro (dans la méthode `d()`) sera propagée jusqu'à la méthode `main()` qui sera alors interrompue avec affichage du nom de l'exception et de l'état de la pile des appels (**Stack-Trace**) :

```
java.lang.ArithmetricException: / by zero
    at javabasic.PropEx.d(PropEx.java:43)
    at javabasic.PropEx.c(PropEx.java:37)
    at javabasic.PropEx.b(PropEx.java:24)
    at javabasic.PropEx.main(PropEx.java:13)
```

Exception in thread "main"

- Il est important de savoir interpréter ces informations de manière à localiser rapidement la cause du problème qui a causé l'interruption de l'application et agir au bon endroit.
- La page suivante décrit comment interpréter cette *Stack-Trace*.

Interprétation de la *Stack-Trace* [2]

- La **Stack-Trace** décrit (dans l'ordre inverse) la séquence (pile) des appels qui ont conduit à l'interruption de l'application.
- La **première ligne** affichée correspond au type d'exception qui a été générée (avec, éventuellement, le texte du message qui lui est associé).
- On trouve ensuite l'enchaînement des invocations de méthodes où chacune des lignes est structurée de la manière suivante :

```
java.lang.ArithmetricException: / by zero
at javabasic.PropEx.d(PropEx.java:43)
at javabasic.PropEx.c(PropEx.java:37)
at javabasic.PropEx.b(PropEx.java:24)
at javabasic.PropEx.main(PropEx.java:13)

Exception in thread "main"
```

...

at Package.Classe.Méthode(Fichier_Source:No_de_ligne)

...

- Dans le cas d'une application, on trouvera sur la **dernière ligne** de la séquence des appels, la méthode **main()** qui constitue le point d'entrée du programme.

Clause finally [1]

- Une clause **finally** peut être ajoutée à une instruction **try / catch**.
- Cette clause définit un bloc d'instructions qui seront exécutées à la fin de l'instruction **try / catch** indépendamment du fait que des exceptions aient été levées ou non. Le bloc **finally** sera exécuté **dans tous les cas**.
- Si aucune exception n'est levée dans le bloc **try** , le bloc **finally** sera exécuté après la dernière instruction du bloc **try**.
- Si une exception est levée dans le bloc **try** et est capturée par un bloc **catch**, le bloc **finally** sera exécuté après la dernière instruction du bloc **catch**.
- Si une exception est levée dans le bloc **try** et n'est pas capturée par un bloc **catch**, le bloc **finally** sera exécuté avant la propagation de l'exception au niveau supérieur.

Clause finally [2]

- Un bloc **finally** est généralement utilisé pour effectuer des opérations de conclusion (fermeture de fichiers, de connexion réseau, de base de données, etc.) qui devraient être effectuées dans tous les cas de figure.

Le bloc **finally** évite donc de devoir placer ces instructions de conclusion dans le bloc **try** et dans tous les blocs **catch**.

- L'utilisation des instructions **break**, **continue**, **return** ou **throw** (dans les blocs **try** ou **catch**) n'empêche pas l'exécution préalable du bloc **finally**.
- Le bloc **finally** est optionnel mais un bloc **try** doit obligatoirement être accompagné d'au moins un bloc **catch** ou d'un bloc **finally** (ou naturellement des deux).

Clause finally [3]

- Exemple d'utilisation de la clause **finally** :

```
try {  
    openFile(f);  
    content= readFile(f);  
    print(content);  
}  
  
catch (InvalidData invdat) {  
    System.out.println("Les données du fichier sont erronées");  
}  
  
catch (PrintError prerr) {  
    System.out.println("Erreur durant l'impression");  
}  
  
finally {  
    closeFile(f); //--- Effectué dans tous les cas  
}
```

Objet exception

- L'objet exception sert principalement de marqueur pour l'événement qui lui est associé.
- Dans un traitement d'exception (bloc `catch`), l'objet exception est transmis et il peut être utilisé pour obtenir plus d'informations concernant l'événement.
- Quelques méthodes que l'on peut utiliser avec les objets exception :

`getMessage()` : retourne un `String` contenant le message (texte) associé à l'exception

`toString()` : retourne un `String` contenant le nom de l'exception suivi du message associé

`printStackTrace()` : affiche sur la console de sortie le nom de l'exception, le message associé ainsi que l'état de la pile des appels qui ont conduit au traitement de l'exception (*Stack-Trace*)

`getStackTrace()` : retourne les informations de la *Stack-Trace* sous forme d'un tableau de `StackTraceElement`

Créer de nouveaux types d'exceptions

- Pour créer ses **propres types d'exceptions**, il suffit de dériver (spécialiser) une classe de type **Throwable** (choisir une des classes existantes proche de celle que l'on souhaite créer ou, sinon, dériver la classe générale **Exception**).
- Généralement, dans cette sous-classe, on crée uniquement deux constructeurs : un constructeur sans paramètre et un constructeur qui prend un message (**String**) en paramètre.
- On crée rarement de nouveaux champs ou de nouvelles méthodes.

```
public class ClockException extends Exception {  
    public ClockException() {  
        super();  
    }  
    public ClockException(String message) {  
        super(message);  
    }  
}
```

A prendre tel quel pour l'instant
(sera étudié ultérieurement)

Utiliser des exceptions personnalisées

- L'utilisation des types d'exceptions que l'on a créés est identique à l'utilisation des types d'exceptions pré-définis.

```
    . . .
if (hour < 0 || hour > 23) {
    throw new ClockException("Heure incorrecte");
}
. . .
```

```
    . . .
try {
    . . .
}
catch (ClockException clockErr) {
    System.out.println(clockErr);
    . . .
}
catch (Exception otherErr) {
    . . .
}
. . .
```



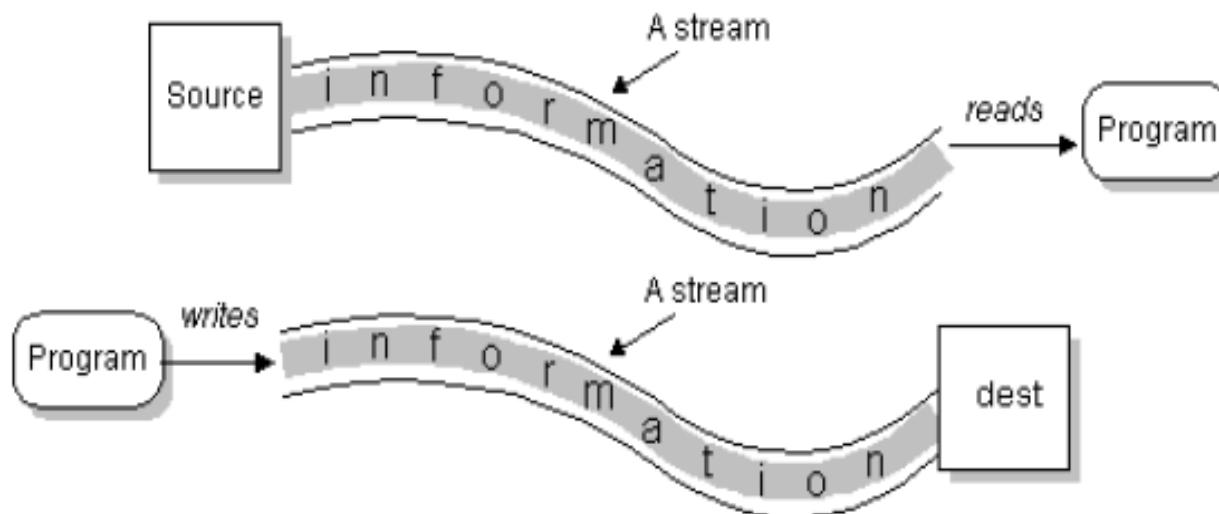
Les entrés/Sorties

Entrées / Sorties (I/O)

- Les **entrées/sorties** (*Input/Output ou I/O*) permettent à un programme de communiquer avec certains périphériques.
- Les entrées/sorties permettent par exemple :
 - de lire et d'écrire sur la console
(lecture au clavier et affichage à l'écran en mode '*caractères*')
 - d'accéder aux fichiers et répertoires des disques
 - de communiquer avec d'autres applications
(localement ou au travers du réseau)
- Les librairies de la plate-forme Java offrent un grand nombre de classes dédiées aux entrées/sorties. La plupart des classes se trouvent dans le package **java.io** (qui comporte plus de 50 classes !).
- Au fil des versions, d'autres packages sont venus compléter la librairie. Il y a notamment le package **java.nio**, le package **java.nio.file**, etc.

Flux (Stream)

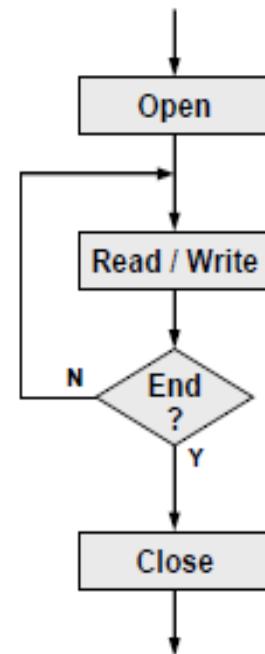
- La notion de **flux (Stream)** caractérise un chemin (une voie) de communication entre une **source** d'information et sa **destination**.



- L'accès aux informations s'effectue de manière séquentielle.
- Les librairies Java distinguent **deux genres de flux** :
 - Les **flux binaires (Byte Stream)** qui peuvent représenter des données quelconques (nombres, données structurées, programmes, sons, images, ...)
 - Les **flux de caractères (Character Stream)** qui représentent des textes (chaînes de caractères) au format *Unicode*.

Utilisation des flux

- La **communication avec un flux** comprend trois phases :
 - L'**ouverture** du flux, en entrée (pour la lecture) ou en sortie (pour l'écriture)
 - La **lecture** ou de **l'écriture** des données (généralement des bytes ou des caractères)
 - La **fermeture** du flux



- Quelques classes importantes pour utiliser les flux courants :

	Flux d'entrée	Flux de sortie
Flux binaires	DataInputStream	DataOutputStream
Flux de caractères	BufferedReader	PrintWriter

La package java.io définit ainsi plusieurs classes

	Flux en lecture	Flux en sortie
Flux de caractères	<code>BufferedReader</code> <code>CharArrayReader</code> <code>FileReader</code> <code>InputStreamReader</code> <code>LineNumberReader</code> <code>PipedReader</code> <code>PushbackReader</code> <code>StringReader</code>	<code>BufferedWriter</code> <code>CharArrayWriter</code> <code>FileWriter</code> <code>OutputStreamWriter</code> <code>PipedWriter</code> <code>StringWriter</code>
Flux d'octets	<code>BufferedInputStream</code> <code>ByteArrayInputStream</code> <code>DataInputStream</code> <code>FileInputStream</code> <code>ObjectInputStream</code> <code>PipedInputStream</code> <code>PushbackInputStream</code> <code>SequenceInputStream</code>	<code>BufferedOutputStream</code> <code>ByteArrayOutputStream</code> <code>DataOutputStream</code> <code>FileOutputStream</code> <code>ObjectOutputStream</code> <code>PipedOutputStream</code> <code>PrintStream</code>

Entrées/Sorties standard (console)

- Les flux suivants sont prédéfinis dans la classe **System**. Ces flux sont toujours ouverts et on ne les ferme pas.
 - **System.in** Entrée standard (lecture du **clavier**) [**InputStream**]
 - **System.out** Sortie standard (affichage à l'**écran**) [**PrintStream**]
 - **System.err** Sortie des messages d'erreur [**PrintStream**]
(souvent identique à **out** par défaut)

- Écriture sur la console :

```
System.out.print("Résultats : ");           //--- Reste sur la même ligne
```

```
System.out.println(5*4 + ", " + 6);
```

```
System.out.println(5*4 + ", " + 4+2);    //--- Attention au piège !
```

```
System.err.println("Erreur : Nom de fichier incorrect");
```

Lecture au clavier

- La lecture au clavier s'effectue en imbriquant (en *enrobant*) le flux standard (**System.in**) dans deux constructeurs de classes qui se chargeront de la conversion des octets en caractères *Unicode* et de la mise en place d'une mémoire tampon intermédiaire.
- Lecture au clavier :

```
BufferedReader keyboard = new BufferedReader(  
    new InputStreamReader(System.in));  
.  
.  
.  
String s = keyboard.readLine();
```

Remarques : La méthode **readLine()** attend jusqu'à ce que l'utilisateur introduise des caractères au clavier et termine la saisie en pressant sur la touche *Return (Enter)*.

Le caractère fin de ligne n'est pas inclus dans le String retourné.

Si l'utilisateur presse uniquement sur *Return*, le String sera vide (longueur = 0).

Exemple de lecture au clavier

```
import java.io.* ;
public class Clavier {
    public static void main(String[] args) {
        try {
            BufferedReader flux = new BufferedReader(
                new InputStreamReader(System.in));
            System.out.print("Entrez votre nom et prénom : ");
            String nom = flux.readLine();
            System.out.println("Bonjour : " + nom);
            flux.close();
        } catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

Lecture au clavier

- Une autre manière de faire (souvent plus pratique) est d'utiliser la classe **Scanner** (`java.util`) qui permet également de saisir mais, en plus, de transformer et d'interpréter ce qui a été saisi au clavier.
- Exemple d'utilisation :

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

System.out.print("Enter an integer (or 'x' to exit) : ");
while (scan.hasNextInt()) {
    int i = scan.nextInt();
    System.out.println("Value = " + i);
    System.out.print("Enter an integer (or 'x' to exit) : ");
}
```

Remarques : La classe **Scanner** offre de nombreuses possibilités pour décomposer et interpréter les informations lues.

Pour plus de détail il faut consulter la documentation (API).

Importation des classes I/O

- La plupart des classes d'entrées/sorties se trouvent dans la librairie (package) `java.io` (ou év. dans `java.nio`).
- Tout programme qui utilise des entrées/sorties (sauf s'il se limite aux flux standard) devra préalablement importer toutes les classes externes nécessaires.
- On utilisera pour cela la pseudo-instruction :

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

- Cette instruction doit être placée avant la déclaration de la classe qui utilise les entrées/sorties (juste après `package ...`).
- Elle a pour effet de **rendre visible** toutes les classes contenues dans le paquetage `java.io` (importation sélective également possible).
- Pour pouvoir utiliser la classe `Scanner` il faut également l'importer :


```
import java.util.Scanner;
```

Traitement des exceptions

- La plupart des opérations d'entrées/sorties risquent de générer des exceptions du type **IOException** (une sous-classe d'**Exception**).
- Ces exceptions (contrôlées) sont déclarées dans la signature des méthodes concernées et le compilateur impose donc au programmeur de les traiter ou de déclarer leur propagation.
- Il faut éviter de mettre des instructions **try / catch** à chaque ligne.
- Grâce au mécanisme de propagation des exceptions, il est possible de concentrer le traitement à un seul endroit (ou à quelques endroits stratégiques judicieusement choisis).
- Dans les applications, il faut trouver le bon compromis et placer les traitements d'exception aux endroits où l'on peut prendre les mesures adéquates et/ou informer correctement l'utilisateur.
 - Pas de message du genre : "Erreur d'entrée/sortie"
 - Mais plutôt un message clair : "Le fichier C:\Image\Tx1.gif n'existe pas"

Hiérarchie de la classe IOException

- La classe **IOException** comporte un grand nombre de sous-classes correspondant à différents types d'événements qui peuvent survenir lorsqu'on travaille avec les entrées/sorties.

```
java.lang.Throwable
  +-- java.lang.Exception
    +-- java.io.IOException
      +-- ChangedCharSetException
      +-- CharacterCodingException
      +-- CharConversionException
      +-- CloseChannelException
      +-- EOFException
      +-- FileNotFoundException
      +-- FileLockInterruptionException
      +-- MalformedURLException
      +-- ProtocolException
      +-- SocketException
      +-- UnknownHostException
      +-- UnsupportedEncodingException
      +-- ZipException
      +-- . . .
```

Plusieurs de ces classes comportent elles-mêmes des sous-classes

Les flots de caractères d'entrée et de sortie

Le paquetage `java.io`

`FileWriter, FileReader` : pour les caractères

`PrintWriter, BufferedReader` : pour les String

FileWriter et PrintWriter

- ❑ On peut diriger un flot de sortie vers un fichier (on choisit le nom du fichier)
 - `FileWriter fw = new FileWriter("fic.txt");`
 - ❑ **FileWriter** permet d'écrire un caractère à la fois
 - `void write(int unicode);`
 - ❑ Pour écrire un **string**, il faut écrire chaque caractère les uns après les autres
 - ❑ Pour écrire un **int**, il faut aussi le décomposer en caractères !
 - ❑ Pas très pratique ? On préfère généralement utiliser un **PrintWriter**:
 - `PrintWriter pw = new PrintWriter(fw);`
 - `pw.print("message");`
- ↑
Obligatoire !

Écrire dans un fichier sur le disque ?

- ❑ Il faut choisir un nom pour le fichier : fic.txt
- ❑ Il faut **ouvrir un flot de sortie** vers le fichier :
 - `FileWriter fw = new FileWriter("fic.txt");`
 - `java.io.FileWriter` est un flot **de caractères**
- ❑ Pour manipuler des lignes plutôt que des caractères :
 - `PrintWriter pw = new PrintWriter(fw);`
 - On est obligé de passer par le `FileWriter` !
- ❑ Pour écrire ?
 - `pw.println("Une ligne");`
- ❑ Il faut **fermer le flot** quand on a fini
 - `pw.close();`
- ❑ Et si il y a une erreur lors de l'ouverture, la fermeture, l'écriture ?
 - Une **Exception** est levée, il FAUT l'attraper !

Et pour les entrées ? System.in

- System.in est le **flot d'entrée standard**

```
int read(byte[] b);
```

- La méthode **read** remplit le tableau **b** avec les octets lus et renvoie le nombre d'octets lu.
- Heureusement, il y a des classes enveloppantes pour traduire les octets en chaînes de caractères : **BufferedReader**.
- On peut aussi utiliser la classe **java.util.Scanner**.

```
try {  
    Scanner sc = new Scanner(System.in);  
    String lu = sc.nextLine();  
    // ou int v = sc.nextInt(); // si on lit un entier  
    // utiliser lu et/ou v ...  
} catch (Exception e) { // obligatoire !  
    System.err.println("Erreur de lecture : " +  
        e.getMessage());  
}
```

Lecture d'un fichier texte

```
import java.io.*;          Ouvre un flot d'entrée de caractères
public class Lecture {
    static public void main(String[] args) {
        try {
            FileReader fr = new FileReader("fic.txt");
            Scanner sc = new Scanner(f); ← Permet le nextLine
            while(sc.hasNextLine()) { // toutes les lignes
                String lu = sc.nextLine(); ← Lit une ligne du flot
                System.out.println("Lu:"+lu); // affiche la ligne
            }
            sc.close(); ← Ferme le flot
        } catch(Exception ex) {
            System.err.println("Erreur de lecture : "+ex);
        }
    }
}
```

Les flux de traitement

InputStreamReader (OutputStreamWriter):

Cette classe permet de transformer un flux de données binaires en un flux de caractères. Elle est très utile lorsque le tampon ou l'encodage par défaut de *FileReader* ne conviennent pas. Elle possède un constructeur qui prend en paramètres un flux d'entrée de données binaires et un *Charset* (objet servant à définir l'encodage à utiliser). La méthode *read()* lit le nombre nécessaire d'octets constituant un caractère.

BufferedReader (BufferedWriter):

Cette classe permet l'emploi d'un tampon (dont on peut spécifier la taille) lors de la lecture d'un flux de caractères. Elle est très utile pour améliorer la performance de l'opération de lecture. Son emploi est analogue à celui de *BufferedInputStream*.

LineNumberReader:

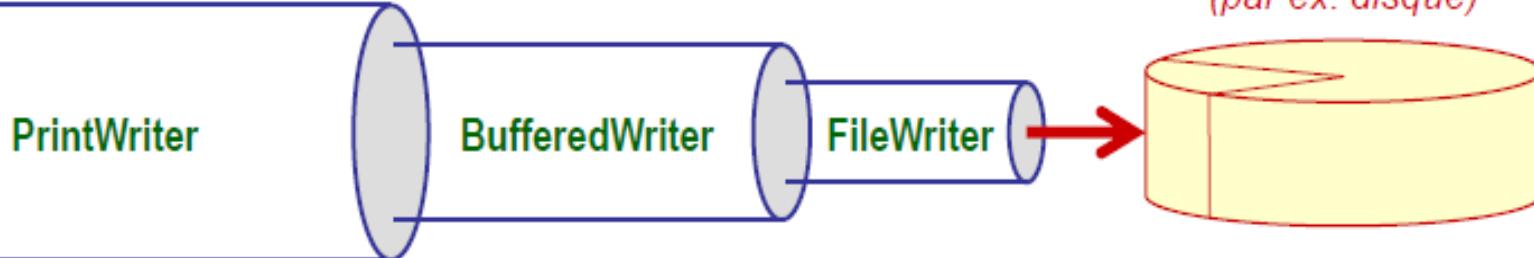
sous-classe de *BufferedReader*, elle en hérite donc l'utilisation d'un tampon et permet en plus de lire ligne par ligne (tout en les comptant) grâce à la méthode *readLine()*.

PrintWriter:

permet d'écrire des caractères formatés, très utile pour l'affichage en mode texte.

Lecture et écriture des flux

- Pour lire un fichier texte il faut ouvrir un flux d'entrée
 - La classe **FileReader** constitue une des couches de base et comporte les méthodes élémentaires pour lire le flux depuis le système de fichiers
- Pour écrire un fichier texte il faut ouvrir un flux de sortie
 - La classe **FileWriter** constitue une des couches de base et comporte les méthodes élémentaires pour écrire le flux sur le système de fichiers
- La lecture et l'écriture de fichiers textes sont réalisées en enveloppant la couche de base avec des outils de plus haut niveau (imbrication)
 - Mémoire tampon (*buffer*) pour minimiser les accès au disque
 - Conversion bytes ⇔ caractères (encodage *Unicode*)
 - Découpage en lignes / Insertion des retours à la ligne, etc.
- Illustration (pour l'écriture) :



Écriture d'un fichier texte

- Pour écrire séquentiellement dans un fichier de texte on peut utiliser la classe **PrintWriter** de la manière suivante:

- Ouverture*

```
String filename = "C:\\Temp\\TestOut.txt";
PrintWriter p = new PrintWriter(
    new BufferedWriter(
        new FileWriter(filename))));
```

- Écriture*

```
p.print("Hello");
p.println(" world" + 3*2);
p.println("Fin du fichier " + filename);
```

- Fermeture*

```
p.close();
```

Lecture d'un fichier texte

- Pour lire séquentiellement le contenu d'un fichier de texte on peut utiliser la classe **BufferedReader** de la manière suivante :

- Ouverture*

```
String filename = "C:\\Temp\\TestIn.txt";
BufferedReader r = new BufferedReader(
    new FileReader(filename));
```

- Lecture* (ligne par ligne)

```
String s;
s = r.readLine();
while (s != null) {      //--- Tant qu'il y a encore des lignes...
    System.out.println(s);
    s = r.readLine();
}
```

- Fermeture*

```
r.close();
```

Lecture d'un fichier texte

- La classe **Scanner** (`java.util`) offre différentes méthodes pour décomposer les données d'un flux de caractères ou d'un **String**.

- *Ouverture*

```
String filename = "D:\\Foo\\Config\\Init.cfg";
Scanner scf = new Scanner(new BufferedReader(
    new FileReader(filename)));
```

- *Lecture* (ligne par ligne, utilisation d'un 2^{ème} Scanner pour analyser chaque ligne)

```
String line;
while (scf.hasNextLine()) {
    line = scf.nextLine();
    Scanner scs = new Scanner(line);
    scs.useDelimiter("=");
    //--- Séparateur (regex pattern)
    String key = scs.next();
    float val = scs.nextFloat();
    ...
}
```

Length=1.2
Width=5.5
Depth=0.27

- *Fermeture*

```
scf.close();
```

Contenu du fichier
Init.cfg

Lecture séquentielle d'un fichier texte

```
import java.io.*;
public class Pg1 {
    public static void main(String [] args) {
        String l, w, filename;
        BufferedReader f, keyboard;
        try {
            keyboard = new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Entrer le nom du fichier : ");
            filename = keyboard.readLine();
            f = new BufferedReader(new FileReader(filename));
            l = f.readLine();
            while (l != null) {
                if (l.charAt(0)=='#')
                    System.out.println("-----");
                else
                    System.out.println(l);
                l = f.readLine();
            }
            f.close();
        }
        catch (IOException e) {
            System.err.println("Erreur: " + e);
        }
    }
}
```

Les flots d'octets d'entrée et de sortie

Le paquetage `java.io`

`FileInputStream` : lecture (entrée)

`FileOutputStream` : écriture (sortie)

Octets ou caractères ?

- En pratique il est rare qu'on sauvegarde les données au format texte
 - Ça prend trop de place !
 - L'entier le plus petit est : -2147483648
 - Il faut 11 caractères pour coder un nombre sur 32 bits !
- Les fichiers courants sont mémorisés en binaire
 - Suite d'octets
 - Difficile à lire par l'homme
 - Prend moins de place, un entier 32 bits prend 4 octets !
 - Exemples : .doc, .bmp, .exe, ...

Lecture et écriture dans un fichier

La lecture et l'écriture dans un fichier s'effectue par le biais de ces deux objets: **FileInputStream** et **OutputStream**

Pour que l'objet **FileInputStream** fonctionne, le fichier doit exister ! Sinon l'exception **FileNotFoundException** est levée.

Par contre, si vous ouvrez un flux en écriture (**OutputStream**) vers un chier inexistant, celui-ci sera créé automatiquement !

Lecture et écriture dans un fichier

La classe File:

Classe modélisant un fichier de manière indépendante du système. Cette classe fournit plusieurs méthodes pour gérer les fichiers et les répertoires (interroger des attributs de fichiers ,droits en lecture, en écriture, renommer ou détruire des fichiers...).

Voir exemple en TP

Ecriture dans un fichier

```
import java.io.* ;
public class EcritureFichier {
    public static void main(String[] args) {
        try {
            File fichier = new File("monFichier.txt");
            FileOutputStream flux = new FileOutputStream(fichier);
            String texte = "Hello World!" ;
            for (int i = 0 ; i < texte.length(); i++) {
                flux.write(texte.charAt(i));
            }
            flux.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Lecture à partir d'un fichier

```
import java.io.* ;
public class LectureFichier {
    public static void main(String[] args) {
        try {
            File fichier = new File("monFichier.txt");
            FileInputStream flux = new FileInputStream(fichier);
            int c ;
            while ((c = flux.read()) > -1) {
                System.out.write(c);
            }
            flux.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



Les flux de traitement

Les flux de traitement

Ils viennent se greffer sur les flux de communication afin d'améliorer la performance de lecture et écriture et de réaliser un traitement sur les données lues. Ils se composent des classes:

BufferedInputStream (BufferedOutputStream):

cette classe permet la lecture (d'écrire) de données à l'aide d'un **tampon**. Lorsqu'elle est instanciée, un tableau d'octets est créé afin de servir de tampon. Ce tableau est redimensionné automatiquement à chaque lecture pour contenir les données provenant du flux d'entrée.

DataInputStream (DataOutputStream):

sert à lire (écrire) des données représentant des types primitifs de Java (int, boolean, double, byte, ...) qui ont été préalablement écrits par un DataOutputStream.

Ainsi cette classe possède des méthodes comme : readInt() , readBoolean() , ...

Les flux de traitement

SequenceInputStream:

permet de concaténer deux ou plusieurs InputStream. La lecture se fait séquentiellement en commençant par le premier et en passant au suivant dès qu'on a atteint la fin du flux courant, tout en appelant sa méthode close() .

ObjectInputStream (ObjectOutputStream):

permet de sérialiser «désérialiser» un objet, c'est-à-dire de restaurer un objet préalablement sauvegardé à l'aide d'un ObjectOutputStream.

Sérialisation des objets:

Dans le jargon des développeurs Java, **Sérialiser** un objet consiste à le convertir en un tableau d'octets, que l'on peut ensuite écrire dans un fichier, envoyer sur un réseau au travers d'une socket etc... Ce mécanisme existe depuis les débuts de l'API Java I/O, et il est très pratique. Il suffit de passer tout objet qui implémente l'interface Serializable à une instance de ObjectOutputStream pour sérialiser un objet. L'interface Serializable n'expose aucune méthode, il marque seulement l'objet comme sérialisable.

Les champs qu'on ne veut pas qu'ils soient sérialisés doivent être marqués avec le mot-clé transient. Cela a pour effet de les retirer du flux sérialisé. Après désérialisation, ces champs seront à null.

Voir exemple en TP

Décomposition (*parsing*) des données lues

- Les données enregistrées dans des fichiers textes sont généralement structurées et nécessitent une phase d'analyse et de décomposition (*parsing*) avant de pouvoir les utiliser.
- La plateforme Java offre différents outils pour faciliter ce travail, par exemple :
 - Classes `StringTokenizer` et `StreamTokenizer`
 - Classe `Scanner`
 - Méthode `split()` de la classe `String`
 - Classes `Pattern` et `Matcher` (package `java.util.regex`)
- La classe `StringTokenizer` est la plus ancienne et, si elle est parfois plus performante, elle est cependant moins flexible que `Scanner` ou `split()` qui se basent sur des *Expressions régulières* (*regex pattern*).

Spécification de Scanner

Constructor Summary (Extract)

Scanner(InputStream source)

Constructs a new Scanner that produces values scanned from the specified input stream.

Scanner(Readable source)

Constructs a new Scanner that produces values scanned from the specified source

Scanner(String source)

Constructs a new Scanner that produces values scanned from the specified string.

Method Summary (Extract)

String	findInLine(String pattern) Attempts to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
boolean	hasNext() Returns true if this scanner has another token in its input.
boolean	hasNext(String pattern) Returns true if the next token matches the pattern constructed from the specified string.
boolean	hasNextBoolean() Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true false".
boolean	hasNextDouble() Returns true if the next token in this scanner's input can be interpreted as a double value using the nextDouble() method.
boolean	hasNextInt() Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the nextInt() method.

Spécification de Scanner

Supplémentaire

Method Summary (Extract)

boolean	hasNextLine() Returns true if there is another line in the input of this scanner.
boolean	hasNext...() Returns true if the next token in this scanner's input can be interpreted as a ...
String	next() Finds and returns the next complete token from this scanner.
String	next(String pattern) Returns the next token if it matches the pattern constructed from the specified string.
boolean	nextBoolean() Scans the next token of the input into a boolean value and returns that value.
double	nextDouble() Scans the next token of the input as a double.
int	nextInt() Scans the next token of the input as a int.
String	nextLine() Advances this scanner past the current line and returns the input that was skipped.
...	next...() Scans the next token of the input as a ...
Scanner	skip(String pattern) Skips input that matches a pattern constructed from the specified string.
Scanner	useDelimiter(String pattern) Sets this scanner's delimiting pattern to a pattern constructed from the specified String.

Spécification de StringTokenizer

Constructor Summary

StringTokenizer(String str)

Constructs a string tokenizer for the specified string.

StringTokenizer(String str, String delim)

Constructs a string tokenizer for the specified string.

StringTokenizer(String str, String delim, boolean returnDelims)

Constructs a string tokenizer for the specified string.

Method Summary (*Extract*)

int	countTokens() Calculates the number of times that this tokenizer's nextToken method can be called (before raising an exception).
boolean	hasMoreElements() Returns the same value as the hasMoreTokens method.
boolean	hasMoreTokens() Tests if there are more tokens available from this tokenizer's string.
Object	nextElement() Returns the same value as the nextToken method, except that its declared return value is Object rather than String.
String	nextToken() Returns the next token from this string tokenizer.
String	nextToken(String delim) Returns the next token in this string tokenizer's string.

Utilisation de StringTokenizer

```
import java.util.StringTokenizer; import java.io.*;
public class Pg2 {
    public static void main(String [] args) {
        String l, w, filename;
        StringTokenizer st;
        BufferedReader f, keyboard;
        try {
            keyboard = new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Entrer le nom du fichier : ");
            filename = keyboard.readLine();
            f = new BufferedReader(new FileReader(filename));
            l = f.readLine();
            while (l != null) {
                st = new StringTokenizer(l);
                while (st.hasMoreTokens()) {
                    System.out.print("(" + st.nextToken() + ")");
                }
                System.out.println();
                l = f.readLine();
            }
            f.close();
        }
        catch (IOException e) {
            System.err.println("Erreur: " + e);
        }
    }
}
```

Méthode split()

- Pour décomposer une chaîne de caractères, il existe également la méthode **split()** qui se trouve dans la classe **String**.
- Pour le découpage de la chaîne, cette méthode se base sur une *expression régulière* dont la syntaxe est définie dans la classe **Pattern** (`java.util.regex`).

```
String str = "Mots, séparés, par, des, virgules";
String[] words = str.split(", ");
for (int i=0; i<words.length; i++) {
    System.out.println(words[i]);
}
```

Mots
séparés
par
des
virgules

Fichiers à accès direct [1]

- La classe **RandomAccessFile** (assez indépendante des autres classes du package **java.io**) permet de lire et d'écrire dans un fichier à des positions arbitraire (accès direct, accès aléatoire).
- Un **curseur (File Pointer)** peut être positionné à un emplacement quelconque et les opérations de lecture/écriture auront lieu à partir de cet endroit (après l'opération, le curseur sera automatiquement positionné après le dernier byte lu ou écrit).
- Lors de l'ouverture du fichier, on indique le **fichier** à traiter ainsi que le **mode d'accès** :
 - "r" : lecture uniquement
 - "rw" : lecture et écriture
- Si le fichier n'existe pas, il sera créé à l'emplacement indiqué (selon les paramètres du constructeur).

Fichiers à accès direct [2]

- Exemple d'utilisation avec un accès en lecture/écriture dans un fichier binaire existant :

Fichiers à accès direct [3]

- La classe **RandomAccessFile** est principalement destinée à la manipulation de données binaires car elle n'inclut pas de méthodes de lecture et d'écriture de chaînes de caractères assurant la conversion au format *Unicode* (seuls les 8 bits de plus faible poids de chaque caractère sont considérés).
- La méthode `readLine()` permet de lire une ligne de texte mais ne fonctionne correctement que pour du texte simple (ASCII / 8 bits).
- La méthode `writeBytes()` permet d'écrire du texte (`String`) mais ne fonctionne correctement que pour du texte simple (ASCII / 8 bits). De plus, il faut s'assurer d'ajouter les caractères de terminaison à la fin de chaque ligne.

Remarque : Les caractères de terminaison de lignes (EOL) dépendent de la plate-forme d'exécution :

```
System.getProperty("line.separator")
```

Arguments de la ligne de commande

- En Java, un programme peut recevoir des **arguments** lors de son lancement (des **paramètres d'exécution** sur la ligne de commande).
- Dans le programme on récupère ces paramètres dans un **tableau de chaînes de caractères** (**String[]**) transmis à la méthode **main()**.

```
public class Echo {  
    public static void main(String[] args) {  
        for (int i=0; i<args.length; i++) {  
            System.out.println(args[i]);  
        }  
        if (args.length == 0)  
            System.out.println("No arguments");  
    }  
}
```

```
prompt> java Echo hello 10 world  
hello  
10  
world
```

```
prompt> java Echo  
no arguments
```

Conversions types primitifs ↔ String

- Dans les opérations d'entrées/sorties avec des flux de caractères (notamment en lecture), des conversions entre types primitifs et **String** sont fréquemment nécessaires.
- Rappel des méthodes de conversion :

		Conversions	
Type primitif	Wrapper	en String	de String
byte	Byte	toString(byte b)	parseByte(String s)
short	Short	toString(short s)	parseShort(String s)
int	Integer	toString(int i)	parseInt(String s)
long	Long	toString(long l)	parseLong(String s)
float	Float	toString(float f)	parseFloat(String s)
double	Double	toString(double d)	parseDouble(String s)

D'autres fonctions de conversion sont également disponibles dans la classe **String**, notamment la méthode **valueOf()** qui permet de convertir en **String** les valeurs de tous les types primitifs.

Utilisation des fonctions de conversion

- Un exemple d'utilisation :

```
public class GetRuntimeParams {  
    public static void main(String[] args) {  
        int      p1 = 0;  
        float    p2 = 0;  
        double   p3 = 0;  
        boolean  p4 = false;  
        try {  
            p1 = Integer.parseInt(args[0]);  
            p2 = Float.parseFloat(args[1]);  
            p3 = Double.parseDouble(args[2]);  
            p4 = Boolean.valueOf(args[3]).booleanValue();  
        }  
        catch (Exception e) {  
            System.err.println("Incorrect Runtime Parameters Types or Values");  
            System.err.println("Should be : int, float, double, boolean");  
            return;  
        }  
        . . .  
        . . .  
    }  
}
```



Les Collections



Type de Donnée Abstrait (T.D.A.)

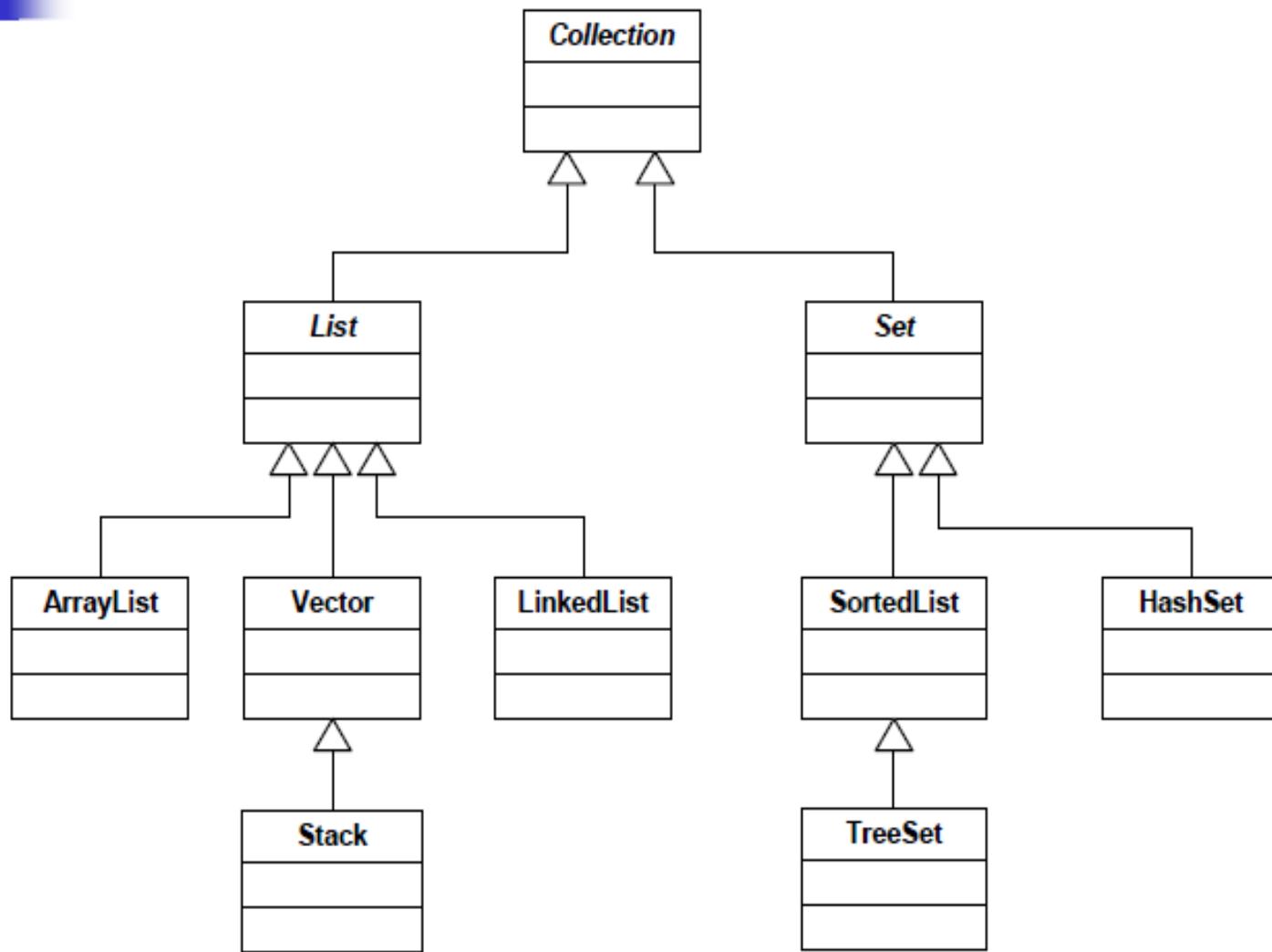
- Un conteneur est un Type de Donnée Abstrait (T.D.A.) (également appelé collection) destiné à contenir des objets.

Il existe deux types de conteneurs :

- Les conteneurs séquentiels :
 - Les vecteurs, piles, files, sacs, ensembles
- Les conteneurs associatifs :
 - Les dictionnaires (map)



Les collections séquentielles

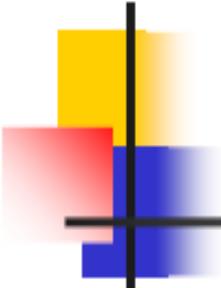




Parcourir les collections

- Tous les conteneurs sauf *Stack*, sont parcourus de la même manière à l'aide d'un itérateur.
- Un itérateur est un objet permettant de faire l'abstraction de la structure de donnée fondamentale sous jacente du conteneur. Ainsi les conteneurs *Vector*, *ArrayList*, *LinkedList* incorporent tous un objet de type *Iterator*.
- Principales méthodes d'un itérateur Java

boolean	hasNext ()	Returns true if the iteration has more elements.
	<u>I</u> next ()	Returns the next element in the iteration.
void	remove ()	Removes the last element returned by the iterator



Exemple de parcours d'une liste à l'aide d'un itérateur

```
List<Complex> l = new LinkedList<Complex>();  
l.add(new Complex(3,5));
```

```
Iterator<Complex> it = l.iterator();  
while (it.hasNext())  
    System.out.println(it.next());
```



Boucle foreach

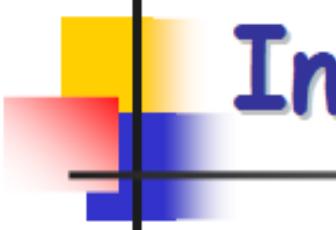
- Java 5 possède une nouvelle construction for (foreach) simplifiant le parcours d'une collection.
Ici `ll` est un objet de type `String`

```
for (String s : ll)  
    System.out.println(s);
```

Suppression d'un élément dans une collection

- Pour supprimer un élément du conteneur il faut l'avoir lu auparavant :

```
it.next();
it.remove();
it.remove(); // faux
it.next(); // correct
it.remove();
it.next();
it.remove();
```



Interface List

- Cette interface déclare les méthodes que doit implémenter *ArrayList*, *Vector*, *LinkedList*

boolean	<u>add (T o)</u> ajoute en fin de liste
T	<u>get (int index)</u>
void	<u>clear ()</u> supprime tous les éléments
boolean	<u>contains (Object o)</u>
boolean	<u>equals (Object o)</u>
boolean	<u>isEmpty ()</u>
<u>Iterator<T></u>	<u>iterator ()</u> Returns un itérateur
boolean	<u>remove (Object o)</u>
int	<u>size ()</u>



Vector, ArrayList, LinkedList

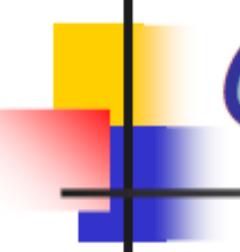
- Ces conteneurs sont des tableaux dynamiques (leur taille augmente automatiquement).
- Ils sont bien adaptés pour manipuler les données situés à un indice quelconque sauf au début ou à la fin de la liste auquel cas il faut leur préférer la liste liée *LinkedList*.
- Toutes ces classes dérivent de l'interface *List*.



Classe *Vector*

- Tableau qui s'agrandi automatiquement.
- La classe est dite synchronisée : l'accès d'un objet de type *Vector* peut être effectué de façon concurrente.
- Exemple d'utilisation :

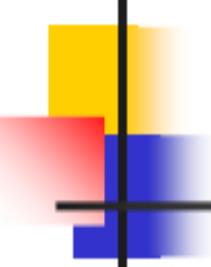
```
List<Complex> l = new Vector<Complex>();  
l.add(new Complex(3, 4));
```



Classe *ArrayList*

- Liste implémentés à l'aide d'un tableau.
- La classe n'est pas synchronisée.
- Exemple d'utilisation :

```
List<Complex> l = new ArrayList<Complex>();  
l.add(new Complex(3, 4));
```



Classe *LinkedList*

- Liste implémentés à l'aide d'une liste liée.
- La classe n'est pas synchronisée.
- Exemple d'utilisation :

```
List<Complex> l = new LinkedList<Complex>();  
l.add(new Complex(3, 4));
```

Exemple complet

```
import java.util.*;
public class TestListe1
{
    public static void main(String[] args)
    {
        List<String> l1 = new ArrayList<String>();
        l1.add("-L3-"); l1.add("-L2-"); l1.add("-L1-"); l1.add("-L0-");
        // Afficher la liste
        System.out.println(l1);
        // Parcourir la liste
        for (Iterator<String> it = l1.iterator(); it.hasNext(); )
            System.out.println(it.next());
        // Trier la liste
        Collections.sort(l1);
        System.out.println(l1);
        // Parcourir la liste avec foreach
        for (String s : l1)
            System.out.println(s);
        //Supprimer le deuxième élément;
        Iterator<String> it = l1.iterator();
        it.next();
        it.next();
        it.remove();
        System.out.println(l1);
    }
}
```

[-L3-, -L2-, -L1-, -L0-]
-L3-
-L2-
-L1-
-L0-
[-L0-, -L1-, -L2-, -L3-]
-L0-
-L1-
-L2-
-L3-
[-L0-, -L2-, -L3-]

Liste : exemple 2

```
public class TestListe
{
    public static void main(String[] args)
    {
        List l = new Vector();
        l.add("-V1-"); l.add("-V2-"); l.add("-V3-");
        System.out.println(l);
        Iterator iv = l.iterator();
        while (iv.hasNext())
            System.out.println(iv.next());

        List l2 = new LinkedList();
        l2.add("-L1-"); l2.add("-L2-"); l2.add("-L3-");
        System.out.println(l2);
        Iterator il = l2.iterator();
        while (il.hasNext())
            System.out.println(il.next());
    }
}
```

Liste : exemple 3

```
LinkedList l2 = new LinkedList();
l2.add("-L1-"); l2.add("-L2-"); l2.add("-L3-");
l2.addFirst("-L0-");
.
```

- On veut insérer, dans l'exemple ci-dessus, si un élément en tête de liste.
On ne peut pas le faire à l'aide de la méthode générale `add` de l'interface `List`, mais avec la méthode `addFirst` de la classe `LinkedList`
 - Justification : `addFirst` n'est pas présente dans l'interface `List` parce que cette opération est très coûteuse pour les structures de données de type vecteur.
- Pour utiliser `addFirst` il faut créer directement un objet du type `LinkedList`. En affichant la liste on obtient

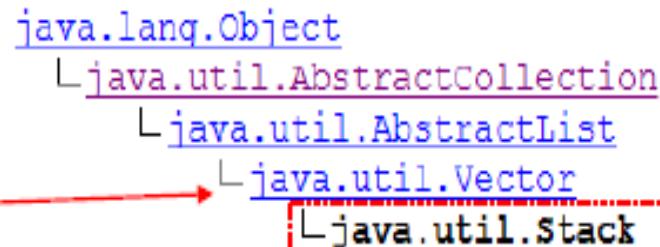
`[-L0-, -L1-, -L2-, -L3-]`

Classe Vector : exemple 4

```
import java.util.Vector;
public class TestVecteur
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        v.add(new Etudiant("Dupond", 2005));
        v.add(new Etudiant("Lajoie", 2005));
        v.add(new Etudiant("Durand", 2006));
        System.out.println(v.toString());
        v.insertElementAt(new Etudiant("Dural", 2007), 2);
        System.out.println(v.toString());
        v.remove(2);
        System.out.println(v.toString());
        Etudiant[] e = new Etudiant[v.size()];
        e = (Etudiant[])v.toArray(e);
        for (int i=0; i < v.size(); i++)
            System.out.println(e[i]);
    }
}
```

Classe Stack

- Une pile est un conteneur séquentiel. Les piles java utilisent une SDF vecteur



Method Summary

<code>boolean</code>	<code>empty()</code>	Tests if this stack is empty.
<code>Object</code>	<code>peek()</code>	Looks at the object at the top of this stack without removing it from the stack.
<code>Object</code>	<code>pop()</code>	Removes the object at the top of this stack and returns that object as the value of this function.
<code>Object</code>	<code>push(Object item)</code>	Pushes an item onto the top of this stack.
<code>int</code>	<code>search(Object o)</code>	Returns the 1-based position where an object is on this stack.



Ensembles HashSet, TreeSet

- *HashSet* et *TreeSet* implémentent l'interface *Set*
- *HashSet* utilise une table d'adressage dispersée, alors que *TreeSet* utilise un arbre binaire.
- L'accès aux éléments d'un ensemble ne se fait pas à l'aide d'un indice et une seule occurrence de donnée est autorisée dans l'ensemble.

Méthodes disponibles dans l'interface Set

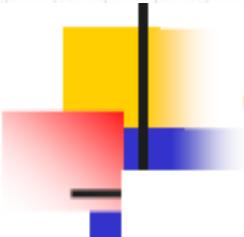
boolean	<u>add(E e)</u>	Adds the specified element to this set if it is not already present (optional operation).
void	<u>clear()</u>	Removes all of the elements from this set (optional operation).
boolean	<u>contains(Object o)</u>	Returns true if this set contains the specified element.
boolean	<u>containsAll(Collection<?> c)</u>	Returns true if this set contains all of the elements of the specified collection.
boolean	<u>equals(Object o)</u>	Compares the specified object with this set for equality.
int	<u>hashCode()</u>	Returns the hash code value for this set.
boolean	<u>isEmpty()</u>	Returns true if this set contains no elements.
boolean	<u>remove(Object o)</u>	Removes the specified element from this set if it is present (optional operation).
boolean	<u>removeAll(Collection<?> c)</u>	Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	<u>retainAll(Collection<?> c)</u>	Retains only the elements in this set that are contained in the specified collection (optional operation).
int	<u>size()</u>	Returns the number of elements in this set (its cardinality).

Exemples utilisant l'interface Set

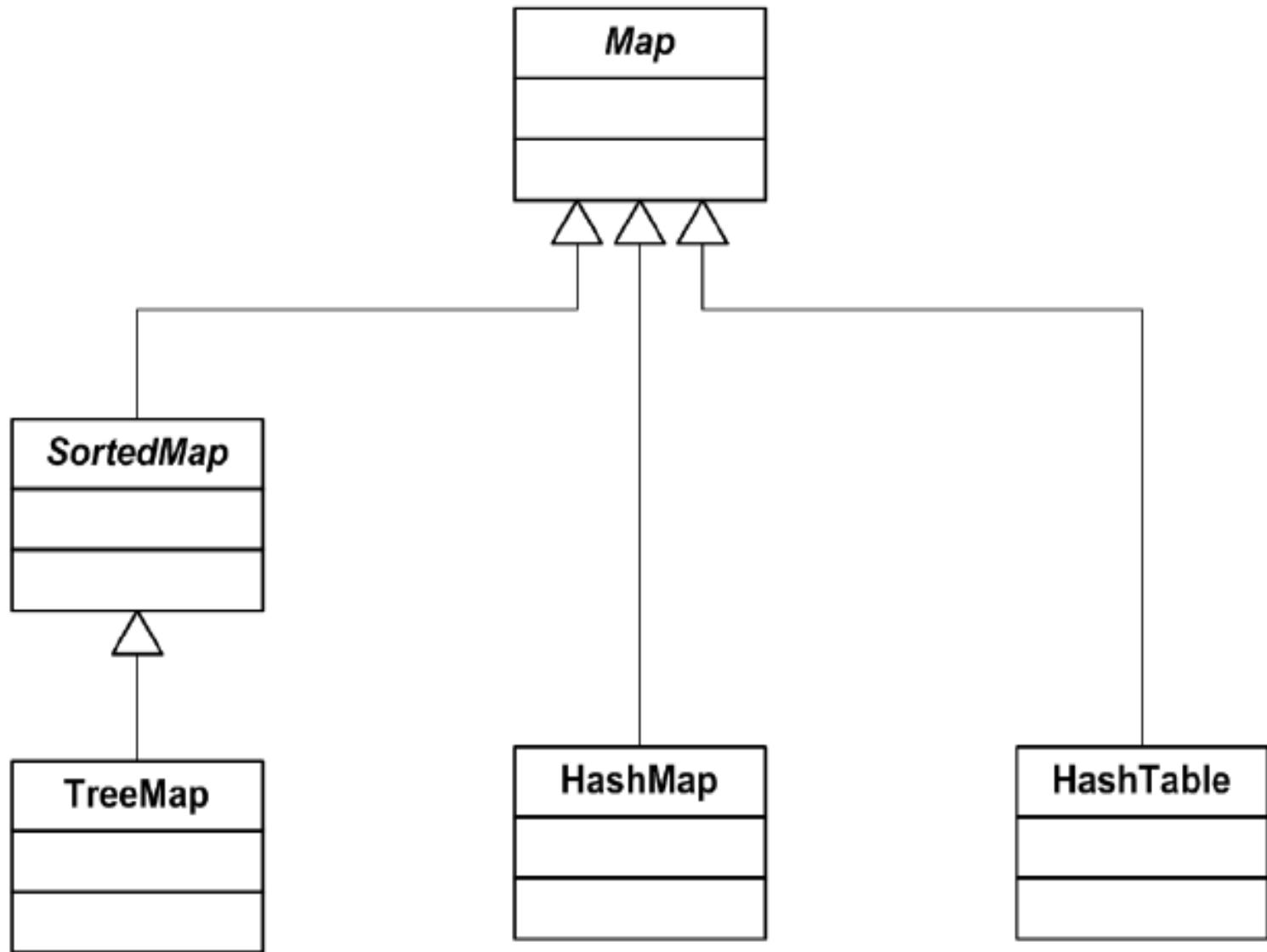
```
import java.util.*;
public class TestSet
{
    public static void main(String[] args)
    {
        System.out.println("TreeSet");
        Set<String> ts = new TreeSet<String>();
        ts.add("-1-"); ts.add("-3-"); ts.add("-2-"); ts.add("-1-");
        System.out.println(ts);
        System.out.println(ts.contains("-2-"));
        System.out.println(ts.contains("-4-"));

        System.out.println("HashSet");
        Set<String> hs = new HashSet<String>();
        hs.add("-1-"); hs.add("-3-"); hs.add("-2-"); hs.add("-1-");
        System.out.println(hs);
        System.out.println(hs.contains("-2-"));
        System.out.println(hs.contains("-4-"));
    }
}
```

Sortie
TreeSet
[-1-, -2-, -3-]
true
false
HashSet
[-3-, -1-, -2-]
true
false



conteneurs associatifs





Conteneurs associatifs

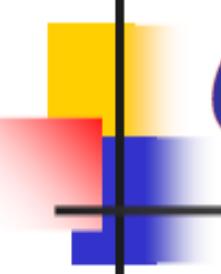
- Les données sont organisées par paires : clé + donnée.

```
Map<String, String> m = new TreeMap<String, String>();  
m.put("Lajoie", "05 46 66 33 50");
```

- "Lajoie" est la clé,
"05 46 66 33 50" la donnée.
- La clé ou index sert à retrouver la donnée associée.

Classe MAP (Interface)

void	<u>clear()</u>	Removes all attributes from this Map.
boolean	<u>containsKey(Object name)</u>	Returns true if this Map contains the specified attribute name (key).
boolean	<u>containsValue(Object value)</u>	Returns true if this Map maps one or more attribute names (keys) to the specified value.
Object	<u>get(Object name)</u>	Returns the value of the specified attribute name, or null if the attribute name was not found.
String	<u>getValue(String name)</u>	Returns the value of the specified attribute name, specified as a string, or null if the attribute was not found.
boolean	<u>isEmpty()</u>	Returns true if this Map contains no attributes.
<u>Set<Object></u>	<u>keySet()</u>	Returns a Set view of the attribute names (keys) contained in this Map.
Object	<u>put(Object name, Object value)</u>	Associates the specified value with the specified attribute name (key) in this Map.
Object	<u>remove(Object name)</u>	Removes the attribute with the specified name (key) from this Map.
int	<u>size()</u>	Returns the number of attributes in this Map.



Classe TreeMap

- Un dictionnaire *TreeMap* trie les clés.
Exemple : annuaire téléphonique utilisant un *TreeMap*

```
Map<String, String> m = new TreeMap<String, String>();  
m.put("Lajoie", "05 46 66 33 50");  
m.put("Dupond", "06 46 66 33 00");  
m.put("Durand", "05 46 66 33 20");  
System.out.println(m.keySet());  
System.out.println(m.values());
```

Affiche :

[Dupond, Durand, Lajoie]

[06 46 66 33 00, 05 46 66 33 20, 05 46 66 33 50]

Manipulation d'un TreeMap

```
// Annuaire inversé
Map<String, String> m3 = new TreeMap<String, String>();
Set v = m.keySet();  
Iterator<String> it = v.iterator();
for (int i = 0; i < m.size(); i++)
{
    String s = it.next();
    m3.put(m.get(s), s);
}
System.out.println(m3.get("05 46 66 33 50"));
```

Un TreeMap ne permet pas de récupérer les clés individuellement (index). Il faut utiliser un ensemble Set

On récupère une clé -> s
puis on récupère la donnée associée à la clé que l'on utilise pour indexer le dictionnaire

Affiche :

Lajoie



Classe *HashMap*

- Ne trie pas les clés mais offre un accès 20% plus rapide que le conteneur précédent.
- On peut reprendre intégralement les exemples précédents en remplaçant *TreeMap* par *HashMap*

```
Map<String, String> m2 = new HashMap<String, String>();
```



Les Threads

Actuellement, toutes les machines, qu'elles soient monoprocesseur ou multiprocesseur, permettent d'exécuter plus ou moins simultanément plusieurs programmes (on parle encore de tâches ou de processus).

Sur les machines monoprocesseur, la simultanéité, lorsqu'elle se manifeste, n'est en fait qu'une illusion : à un instant donné, un seul programme utilise les ressources de l'unité centrale ; mais l'environnement "passe la main" d'un programme à un autre à des intervalles de temps suffisamment courts



Java présente l'originalité d'appliquer cette possibilité de multiprogrammation au sein d'un même programme dont on dit alors qu'il est formé de plusieurs *threads* indépendants. Le contrôle de l'exécution de ces différents threads (c'est-à-dire la façon dont la main passe de l'un à l'autre) se fera alors, au moins partiellement, au niveau du programme lui-même et ces threads pourront facilement communiquer entre eux et partager des données.

Il existe deux façons de créer des threads : soit en exploitant la classe pré définie *Thread*, soit en créant une classe spécifique implémentant l'interface *Runnable*.

En Java, un thread est un objet d'une classe qui dispose d'une méthode nommée *run* qui sera exécutée lorsque le thread sera démarré.

Exemple introductif

Voici un programme qui va lancer trois threads simples, chacun d'entre eux se contentant d'afficher un certain nombre de fois un texte donné, à savoir :

- 10 fois "bonjour" pour le premier thread,**
- 12 fois "bonsoir" pour le deuxième thread,**
- 5 fois un changement de ligne pour le troisième thread.**

```
class Ecrit extends Thread
{ public Ecrit (String texte, int nb, long attente)
    { this.texte = texte ; this.nb = nb ;
      this.attente = attente ;
    }
    public void run ()
    { try
        { for (int i=0 ; i<nb ; i++)
            { System.out.print (texte) ;
              sleep (attente) ;
            }
        }
        catch (InterruptedException e) {} // impose par sleep
    }
    private String texte ;
    private int nb ;
    private long attente ;
}
```

```
public class TstThrl
{ public static void main (String args[ ] )
{ Ecrit e1 = new Ecrit ("bonjour ", 10, 5) ;
  Ecrit e2 = new Ecrit ("bonsoir ", 12, 10) ;
  Ecrit e3 = new Ecrit ("\n", 5, 15) ;
  e1.start () ;
  e2.start () ;
  e3.start () ;
}
}
```

bonjour bonsoir bonjour
bonsoir bonjour bonjour bonsoir bonjour
bonjour bonsoir bonjour
bonjour bonsoir bonjour
bonjour bonsoir bonsoir
bonsoir bonsoir bonsoir bonsoir bonsoir

Remarques

Un programme comporte toujours au moins un thread dit "thread principal" correspondant tout simplement à la méthode *main*.

Si l'on appelait directement la méthode *run* de nos objets threads, le programme fonctionnerait mais l'on n'aurait plus affaire à trois threads différents.

Les appels de *sleep* autoriseraient l'environnement à exécuter éventuellement d'autres threads.



La méthode *start* ne peut être appelée qu'une seule fois pour un objet thread donné. Dans le cas contraire, on obtiendra une exception *IllegalThreadStateException*.

La méthode *sleep* est en fait une méthode statique (de la classe *Thread*) qui met en sommeil le thread en cours d'exécution. Nous aurions pu remplacer l'appel *sleep (attente)* par *Thread.sleep (attente)*.

Si nous ne prévoyons pas d'appel de *sleep* dans notre méthode *run*, le programme fonctionnera encore mais son comportement dépendra de l'environnement.

Utilisation de l'interface Runnable

pour créer des threads, vous disposez d'une seconde démarche basée non plus sur une classe dérivée de *Thread*, mais simplement sur une classe implémentant l'interface *Runnable*, laquelle comporte une seule méthode nommée *run*.

Exemple2: Reproduction de l'exemple précédent

```
class Ecrit implements Runnable
{ public Ecrit (String texte, int nb, long attente)
    { this.texte = texte ;
      this.nb = nb ;
      this.attente = attente ;

    public void run ()
    { try
        { for (int i=0 ; i<nb ; i++)
            { System.out.print (texte) ;
              Thread.sleep (attente) ; // attention Thread.sleep
            }
        }
      catch (InterruptedException e) {} // impose par sleep
    }

    private String texte ;
    private int nb ;
    private long attente ;
}
```

```
public class TstThr3
{ public static void main (String args[ ] )
    { Ecrit e1 = new Ecrit ("bonjour ", 10, 5) ;
      Ecrit e2 = new Ecrit ("bonsoir ", 12, 10) ;
      Ecrit e3 = new Ecrit ("\n", 5, 15) ;
      Thread t1 = new Thread (e1) ; t1.start() ;
      Thread t2 = new Thread (e2) ; t2.start() ;
      Thread t3 = new Thread (e3) ; t3.start() ;
    }
}
```

bonjour bonsoir
bonjour bonsoir
bonjour bonjour bonsoir bonjour
bonjour bonsoir bonjour bonjour bonsoir
bonjour bonjour bonsoir
bonsoir bonsoir bonsoir bonsoir bonsoir

Interruption d'un thread

Dans les exemples précédents, les threads s'achevaient tout naturellement avec la fin de l'exécution de leur méthode run.

Dans certains cas, on peut avoir besoin d'interrompre prématulement un thread depuis un autre thread. Ce besoin peut devenir fondamental dans le cas de ce que nous nommerons des "threads infinis".

Java dispose d'un mécanisme permettant à un thread d'en interrompre un autre. La méthode *interrupt* de la classe *Thread* demande à l'environnement de positionner un indicateur signalant une demande d'arrêt du thread concerné

il est possible de connaître l'état de cet indicateur à l'aide de la méthode statique *interrupted* (il existe également *isInterrupted*).

Thread 1

```
t.interrupt() ; // positionne un  
                // indicateur dans t
```

TP: Reproduire le TP précédent,
avec cette fois, les threads sont
"infinis",

```
bonjour bonsoir bonjour bonsoir wbonjour  
bonjour bonsoir
```

```
*** Arret premier thread ***  
bonsoir bonsoir bonsoir  
bonsoir bonsoir  
bonsoir bonsoir  
bonsoir bonsoir  
xbonsoir bonsoir  
bonsoir bonsoir  
bonsoir
```

```
*** Arret deux derniers threads ***
```

Thread 2 nommé t

```
run  
{ .....  
    if (interrupted)  
    { .....  
        return ; // fin du thread  
    }  
}
```

Solution!!

Threads démons et arrêt brutal



Coordination de threads

Dans certains cas, il faudra éviter que deux threads puissent accéder (presque) en même temps au même objet. Ou encore, un thread devra attendre qu'un autre ait achevé un certain travail sur un objet avant de pouvoir lui-même poursuivre son exécution.

Le premier problème est réglé par l'emploi de méthodes synchronisées.

Le second problème sera réglé par des mécanismes d'attente et de notification à l'aide des méthodes *wait* et *notify*.

Méthodes synchronisées

Java permet de déclarer des méthodes avec le mot-clé `synchronized`. À un instant donné, une seule méthode ainsi déclarée peut être appelée pour un objet donné.

Exemple:

Nous allons donc partager deux informations (n et son carré `carre`) entre deux threads. Le premier incrémente n et calcule son carré dans `carre` ; le second thread se contente d'afficher le contenu de `carre`.

Ici, les informations sont regroupées dans un objet `nomb` de type `Nombres`. Cette classe dispose de deux méthodes mutuellement exclusives (`synchronized`) :

calcul qui incrémente *n* et calcule la valeur de *carre*,

- *affiche* qui affiche les valeurs de *n* et de *carre*.

Nous créons deux threads de deux classes différentes :

- *calc* de classe *ThrCalc* qui appelle, à son rythme (défini par appel de *sleep*), la méthode *calcul* de *nomb*,
- *aff* de classe *ThrAff* qui appelle, à son rythme (choisi volontairement différent de celui de *calc*), la méthode *affiche* de *nomb*.

Les deux threads sont lancés par *main* et interrompus lorsque l'utilisateur le souhaite (en frappant un texte quelconque).

Voir TP!!

Notion de verrou

Le verrou (ou une clé) est attribué à la méthode synchronisée appelée pour l'objet et il est restitué à la sortie de la méthode. Tant que le verrou n'est pas restitué, aucune autre méthode synchronisée ne peut le recevoir (bien sûr, les méthodes non synchronisées peuvent, quant à elles, accéder à tout moment à l'objet).

Ce mécanisme d'exclusion mutuelle est basé sur l'objet lui-même et non sur le thread.

L'instruction synchronized

Une méthode synchronisée acquiert donc le verrou sur l'objet qui l'a appelée (implicitement) pour toute la durée de son exécution. L'utilisation d'une méthode synchronisée comporte deux contraintes :

- l'objet concerné (celui sur lequel elle acquiert le verrou) est nécessairement celui qui l'a appelée,
- l'objet est verrouillé pour toute la durée de l'exécution de la méthode.

L'instruction *synchronized* permet d'acquérir un verrou sur un objet quelconque (qu'on cite dans l'instruction) pour une durée limitée à l'exécution d'un simple bloc :

```
synchronized (objet)
{ instructions
}
```

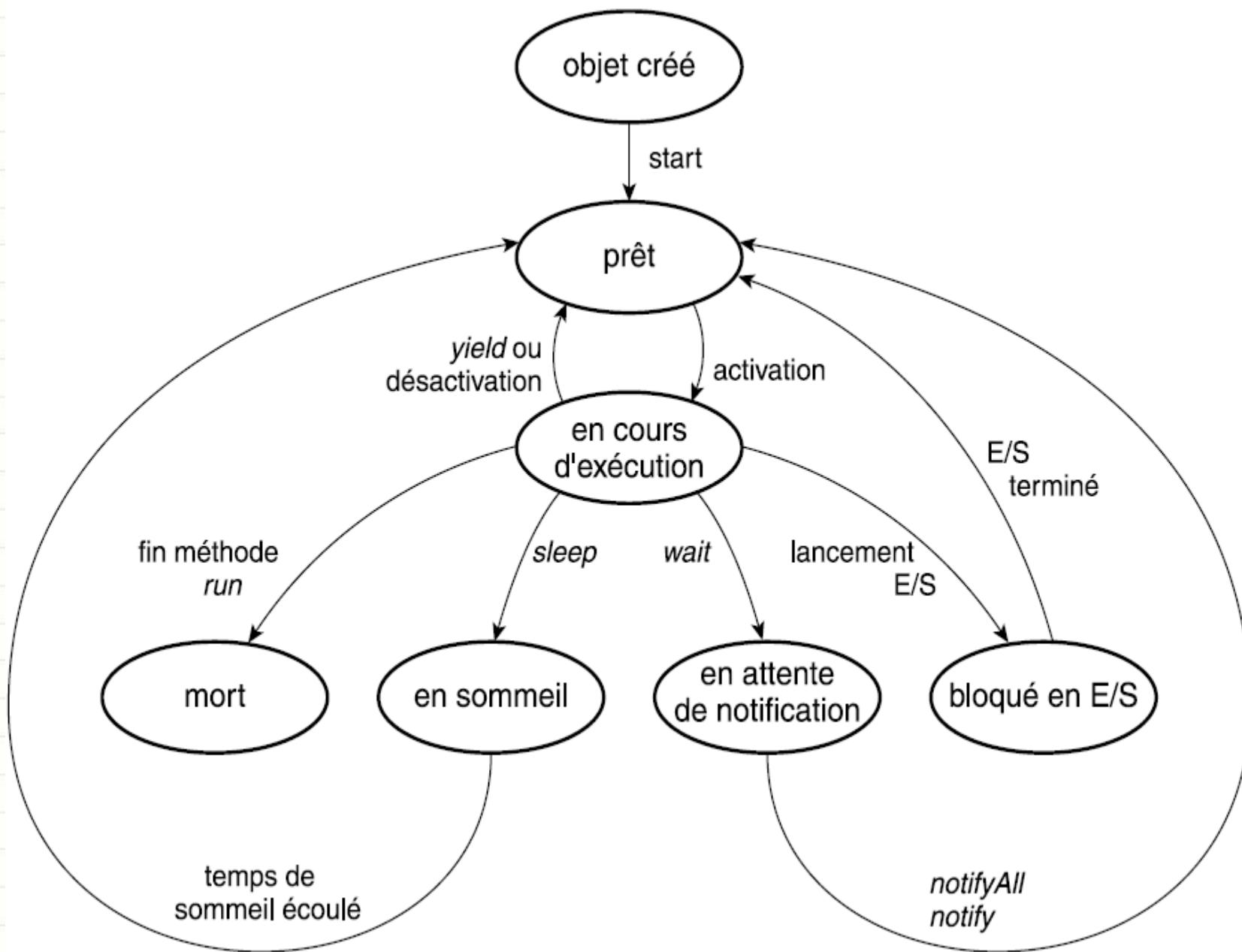
Attente et notification

Il arrive que l'on ait besoin de coordonner l'exécution de threads, un thread devant attendre qu'un autre ait effectué une certaine tâche pour continuer son exécution.

Java offre un mécanisme basé sur l'objet et sur les méthodes synchronisées que nous venons d'étudier :

- une méthode synchronisée peut appeler la méthode *wait* de l'objet dont elle possède le verrou, ce qui a pour effet :
 - de rendre le verrou à l'environnement qui pourra, le cas échéant, l'attribuer à une autre méthode synchronisée,
 - de mettre "en attente" le thread correspondant ; plusieurs threads peuvent être en attente sur un même objet ; tant qu'un thread est en attente, l'environnement ne lui donne pas la main ;
- une méthode synchronisée peut appeler la méthode *notifyAll* d'un objet pour prévenir tous les threads en attente sur cet objet et leur donner la possibilité de s'exécuter. ***Exemples en TP***

Les différents états d'un thread





Les Enumérations

Les Enumérations

Les énumérations constituent une notion nouvelle depuis Java 5. Ce sont des structures qui définissent une liste de valeurs possibles. Cela vous permet de créer des types de données personnalisés.

Par exemple construire le type Langage qui ne peut prendre qu'un certain nombre de valeurs : JAVA, PHP, C, etc.

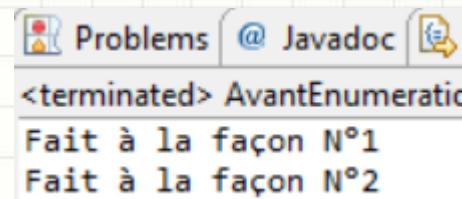
Les Enumérations

les énumérations ne sont pas très difficiles à utiliser et nos programmes y gagnent en rigueur et en clarté.

- Une énumération est une classe contenant une liste de sous-objets.
- Une énumération se construit grâce au mot clé enum.
- Les enum héritent de la classe java.lang.Enum.
- Chaque élément d'une énumération est un objet à part entière.
- Vous pouvez compléter les comportements des objets d'une énumération en ajoutant des méthodes.

Avant les énumérations

```
1 public class AvantEnumeration {  
2  
3     public static final int PARAM1 = 1;  
4     public static final int PARAM2 = 2;  
5  
6     public void fait(int param){  
7         if(param == PARAM1)  
8             System.out.println("Fait à la façon N°1");  
9         if(param == PARAM2)  
10            System.out.println("Fait à la façon N°2");  
11     }  
12  
13    public static void main(String args[]){  
14        AvantEnumeration ae = new AvantEnumeration();  
15        ae.fait(AvantEnumeration.PARAM1);  
16        ae.fait(AvantEnumeration.PARAM2);  
17        ae.fait(4);  
18    }  
19 }
```



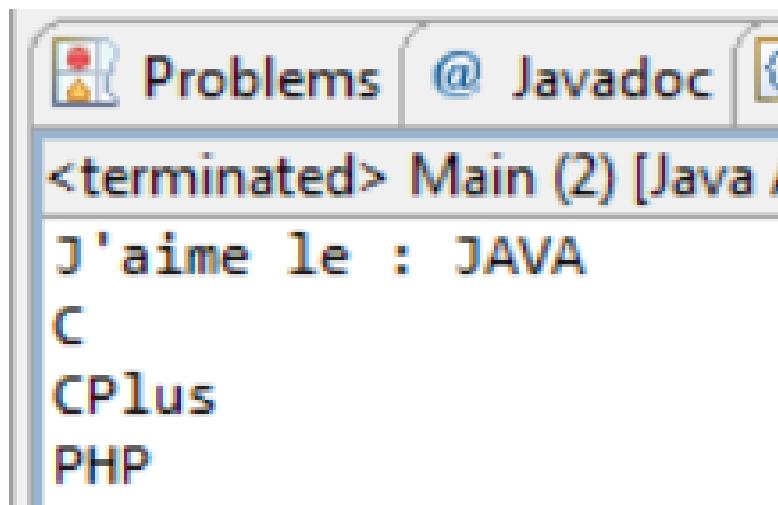
Les Enumérations

Une énumération se déclare comme une classe,
mais en remplaçant le mot-clé `class` par `enum`.
les énumérations héritent de la classe `java.lang.Enum`.

```
1 public enum Langage {  
2     JAVA,  
3     C,  
4     CPlus,  
5     PHP;  
6 }
```

Les Enumérations

```
1 public class Main {  
2     public static void main(String args[]){  
3         for(Langage lang : Langage.values()){  
4             if(Language.JAVA.equals(lang))  
5                 System.out.println("J'aime le : " + lang);  
6             else  
7                 System.out.println(lang);  
8         }  
9     }  
10 }
```



The screenshot shows the Eclipse IDE interface with the 'Problems' view open. The status bar at the bottom displays the message '<terminated> Main (2) [Java Application]'. The output window lists the following text:
J'aime le : JAVA
C
CPlus
PHP

Les Enumérations

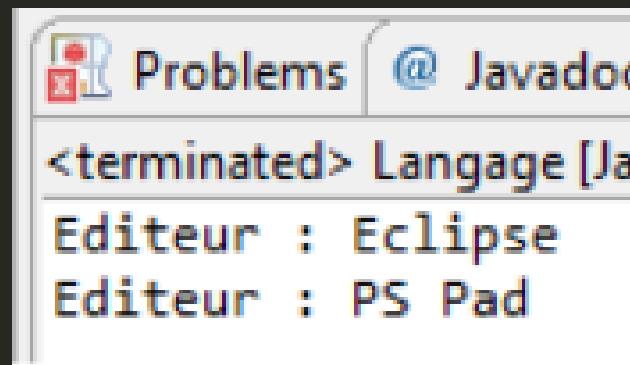
```
1 public enum Langage {  
2     //Objets directement construits  
3     JAVA ("Langage JAVA"),  
4     C ("Langage C"),  
5     CPlus ("Langage C++"),  
6     PHP ("Langage PHP");  
7  
8     private String name = "";  
9  
10    //Constructeur  
11    Langage(String name){  
12        this.name = name;  
13    }  
14  
15    public String toString(){  
16        return name;  
17    }  
18 }
```

Utilisation d'un constructeur avec une enum

```
Problems @ Javadoc Dec  
<terminated> Main (2) [Java Application]  
J'aime le : Langage JAVA  
Langage C  
Langage C++  
Langage PHP
```

Exemples complets :

```
1 public enum Langage {
2     //Objets directement construits
3     JAVA("Langage JAVA", "Eclipse"),
4     C ("Langage C", "Code Block"),
5     CPlus ("Langage C++", "Visual studio"),
6     PHP ("Langage PHP", "PS Pad");
7
8     private String name = "";
9     private String editor = "";
10
11    //Constructeur
12    Langage(String name, String editor){
13        this.name = name;
14        this.editor = editor;
15    }
16
17    public void getEditor(){
18        System.out.println("Editeur : " + editor);
19    }
20
21    public String toString(){
22        return name;
23    }
24
25    public static void main(String args[]){
26        Langage l1 = Langage.JAVA;
27        Langage l2 = Langage.PHP;
28        ...
29        l1.getEditor();
30        l2.getEditor();
31    }
32 }
```



- **Enumération** : ensemble de constantes

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
    THURSDAY, FRIDAY, SATURDAY  
}
```

- Autre exemple : les points cardinaux

```
public enum Pc {  
    NORTH, EST, SOUTH, WEST  
}
```

- Autre exemple : les planètes du système solaire

```
public enum Planet {  
    MERCURY, EARTH, MARS, JUPITER, SATURN,  
    URANUS, NEPTUNE  
}
```

```
public class TellHowDayls {  
    Day day;  
    public TellHowDayls(Day d) {day = d; }  
    public void tellItLikeItIs() {  
        switch (day) {  
            case MONDAY:  
                System.out.println("Mondays are bad.");  
                break;  
            case SATURDAY:  
            case SUNDAY:  
                System.out.println("Weekends are best.");  
                break;  
            default:  
                System.out.println("Midweek days  
                    are so-so.");  
                break;  
        }  
    }  
}
```

Output - Ennum (run) ✘

	run:
	Mondays are bad.
	Weekends are best.
	BUILD SUCCESSFUL (total time: 2 seconds)

```
public class TestEnum {  
    public static void main(String[] args) {  
        TellHowDayls firstDay =  
            new TellHowDayls(Day.MONDAY);  
        firstDay.tellItLikeItIs();  
  
        TellHowDayls LastDay =  
            new TellHowDayls(Day.SUNDAY);  
        LastDay.tellItLikeItIs();  
    }  
}
```

```

public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    EARTH (5.976e+24, 6.37814e6),
    MARS (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7);

    private final double mass;
    private final double radius;
    Planet(double m, double r) {
        mass = m;
        radius = r;
    }
    // universal gravitational cst
    public static final double G = 6.673E-11;
    double surfaceGravity() {
        return G * mass / (radius * radius); }
    double surfaceWeight(double val) {
        return val * surfaceGravity(); }
}

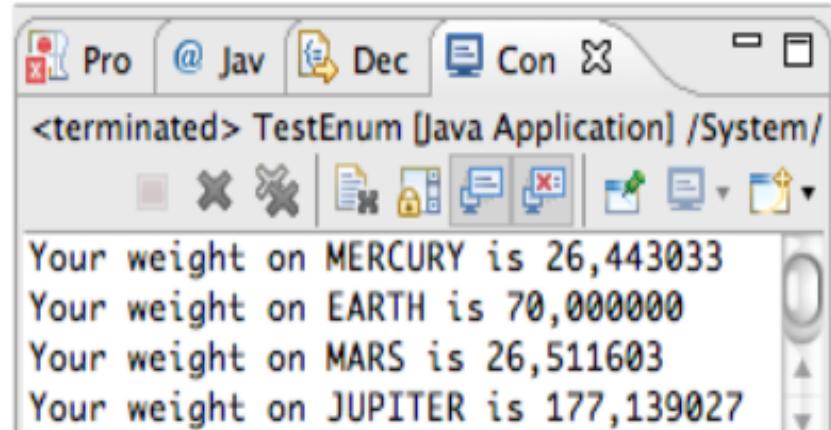
```

```

public class TestEnum {
    public static void main(String[] args) {
        double earthWeight = 70;
        Planet Terre = Planet.EARTH;
        double mass = earthWeight
            /Terre.surfaceGravity();

        for (Planet p : Planet.values())
            System.out.printf(" weight on "
                + p.name() + " is "
                + p.surfaceWeight(mass)
                + "\n");  }
}

```





FIN DE LA PARTIE 3