La programmation orientée objet 4+ 1GIN

Java™

Version 2022/2023

Sources

- *Introduction à Java, Robert Fisch, 2009*
- *Introduction à la programmation,* Robert Fisch, 11TG/T0IF, 2006
- *Programmation en Delphi*, Fred Faber, 12GE/T2IF, 1999
- *Programmation en Delphi*, Fred Faber, 13GI/T3IF, 2006

Rédacteurs

- Fred Faber

Site de référence

- **http://java.cnpi.lu**

Autres sites intéressants

- **http://java.developpez.com/**
- **http://www.javabuch.de**

Table des matières

A. Exercices de révision

… Voir liste des exercices ...

B. Les chaînes de caractères

Le présent chapitre est dédié à la manipulation simple des chaînes de caractères en Java. Il s'agit d'une étude approfondie de la matière vue en classe de 2e. La classe **`String`** est définie dans le paquet **`java.lang`**.

B.1. Les chaînes de caractères « String »

Les chaînes de caractères en Java (par exemple **`"abc"`**), sont représentées comme des instances de la classe **`String`**, qui est une classe spéciale.

B.1.1. Déclaration et affectation – *rappel 2e*

Voici des exemples de déclaration d'une chaîne de caractères :

```
private String name;                        // déclaration d'un champ
private String name = new String("René");   // ... avec initialisation
private String name = "René";               // ... et init. avec une constante

String filename;                // déclaration d'une variable
String filename = "test.doc";   // ... et initialisation avec une constante
```

Comme on peut le voir dans les exemples, les chaînes de caractères en Java sont comprises entre des guillemets (doubles).

B.1.2. Comparaisons – *rappel 2e*

Pour comparer des chaînes de caractères, on **ne** peut **pas** se servir des opérateurs de comparaison standard puisque ces derniers s'appliquent uniquement aux types primitifs (**`double`**, **`int`**, …). Pour les chaînes de caractères, il faut se servir des méthodes suivantes :

Syntaxe	Explications
`boolean equals(String)`	Retourne **`true`** si les chaînes de caractères sont égales, **`false`** sinon.
`int compareTo(String)`	Retourne un nombre positif si cette chaîne est lexicographiquement plus grande que celle passée en tant que paramètre, un nombre négatif sinon. La valeur 0 est retournée si les deux chaînes sont égales.

Exemples

```
String name1 = "abba";
String name2 = "queen";
String name3 = name2;
name1.equals(name2)           =>    false
name1.compareTo(name2)        =>    -16   (nombre négatif car "abba" < "queen")
name2.compareTo(name3)        =>    0     (zéro car "queen" = "queen")
```

B.1.3. Test de contenance

Syntaxe	Explications
`boolean contains(String)`	Retourne **`true`** si la chaîne fournie comme paramètre fait partie de la chaîne actuelle, **`false`** sinon.
`int indexOf(String)`	Retourne la position de la première occurrence de la chaîne de caractères passée en tant que paramètre. Si celle-ci n'est pas contenue dans la chaîne de caractère de base, la valeur -1 est retournée.

Exemples

```
String someText = "Hello world";
System.out.println(someText.indexOf ("Hello"));      // affiche "0"
System.out.println(someText.contains("Hello"));      // affiche "true"
System.out.println(someText.indexOf ("hello"));      // affiche "-1"
System.out.println(someText.contains("hello"));      // affiche "false"
System.out.println(someText.indexOf ("world"));      // affiche "6"
```

Notez, que le premier caractère d'une chaîne de caractères se trouve à la position zéro !

B.1.4. Longueur

Syntaxe	Explications
`int length()`	Retourne la longueur du texte.

Exemples

```
String someText = "Hello world";
System.out.println(someText.length());          // affiche "11" car someText
                                                // contient 11 symboles
```

B.1.5. Copie d'une sous-chaîne

Syntaxe	Explications
`String substring(beginIndex)` `String substring(beginIndex,endIndex)`	Retourne une copie d'une sous-chaine d'un texte donné. **`beginIndex`** est la position de début tandis que **`endIndex`** est la position de fin. **Le caractère à la position `endIndex` n'est plus copié.** Si **`endIndex`** est omis, tous les symboles de **`beginIndex`** jusqu'à la fin sont copiés et retournés.

Exemples

```
String someText = "Hello world";
System.out.println(someText.substring(3));          // affiche "lo world"
System.out.println(someText.substring(6,8));        // affiche "wo"
System.out.println(someText.substring(1,3));        // affiche "el"
System.out.println(someText.substring(0));          // affiche "Hello world"
System.out.println(someText.substring(1));          // affiche "ello world"
```

B.1.6. Conversion en minuscules / majuscules

Syntaxe	Explications
`String toLowerCase()`	Retourne une copie modifiée du texte dans laquelle toutes les lettres ont été transformées en des lettres minuscules.
`String toUpperCase()`	Retourne une copie modifiée du texte dans laquelle toutes les lettres ont été transformées en des lettres majuscules.

Exemples

```
String someText = "Hello world";
System.out.println(someText.toLowerCase());          // affiche "hello world"
System.out.println(someText.toUpperCase());          // affiche "HELLO WORLD"
```

B.1.7. Conversions de types – *rappel 2e*

Souvent on a besoin de convertir des chaînes de caractères en des nombres, respectivement des nombres en des chaînes de caractères. Le méthode la plus simple est d'utiliser la méthode **`valueOf(...)`** des classes **`Integer`**, **`Double`** et **`String`**.

Syntaxe	Explications
`String.valueOf(int)` `String.valueOf(double)`	Convertit un nombre entier ou un nombre décimal en une chaîne de caractères.
`Integer.valueOf(String)`	Convertit une chaîne de caractères en un nombre entier.
`Double.valueOf(String)`	Convertit une chaîne de caractères en un nombre décimal.

Si la conversion d'un texte en un nombre ne réussit pas, une exception du type **`NumberFormatException`** est levée et l'exécution de la méthode concernée se termine.

Exemples

```
String integerRepresentation = "3";
String doubleRepresentation  = "3.141592";

// faire la conversion d'une chaîne de caractères en un nombre entier
int myInt = Integer.valueOf(integerRepresentation);

// faire la conversion d'une chaîne de caractères en un nombre décimal
double myDouble = Double.valueOf(doubleRepresentation);

// faire la conversion d'un nombre entier en une chaîne de caractères
String intText = String.valueOf(myInt);

// faire la conversion d'un nombre décimal en une chaîne de caractères
String doubleText = String.valueOf(myDouble);
```

B.1.8. Autres méthodes intéressantes :

Recherchez dans JavaDoc (**`java.lang.String`**) les méthodes suivantes et remplissez le tableau ci-dessous :

Syntaxe	Explications
`String replace(char,char)`	
`String trim()`	
`char charAt(int)`	
`String[] split(String)`	

B.2. Les types char et String

- Les constantes du type char sont notés entre apostrophes, p.ex : **`'A'`**
  Il n'existe **pas** de constante char vide.
- Une donnée du type **`char`** peut être affectée à une variable du type **`String`**, mais pas inversement (=> employer **`String.charAt(...)`**).
- En Java, les caractères sont codés en Unicode. On peut obtenir le code d'un caractère par un casting en **`int`**, p.ex : **`int charCode = (int)'A';`**
- Inversement, on peut obtenir le caractère correspondant à un certain code par un casting en **`char`**, p.ex : **`char c = (char)65;`**
- Si on compare des caractères, on compare en effet leurs codes.

C. Exceptions

Analysons l'exemple suivant d'une méthode qui calcule le quotient de deux nombres :

```
public class Fraction
{
     ...

     public double getDecimalValue()
     {
          if (denominator!=0)
               return (double)numerator/denominator;
          else
               <???>
     }
}
```

Que faire dans le cas où le dénominateur est égal à zéro? La fonction doit retourner un « **double** » mais pour le cas où « **b==0** », on ne peut pas calculer le quotient « **a/b** »!

C.1. Génération d'une exception

Dans le cas décrit précédemment, la solution consiste à lancer ce qu'on appelle en Java une **exception**. En fait, ce n'est rien d'autre que la production contrôlée d'une erreur.

Solution

```
     public double getDecimalValue()
     {
          if (denominator!=0)
               return (double)numerator/denominator;
          else
               throw new ArithmeticException("Division by zero");
     }
```

Lors du lancement d'une exception, il faut créer une nouvelle instance de ce type d'exception en faisant appel à son constructeur. Normalement, les exceptions possèdent (en plus du constructeur par défaut) un constructeur avec un paramètre du type **String**. Dans ce paramètre, on peut indiquer le message à afficher lors de la génération de l'exception.

Bien qu'il existe un certain nombre d'**exceptions prédéfinies**[1] en Java, il est aussi possible de définir ses propres **exceptions** (→ voir chapitre C.9).

1 http://java.sun.com/javase/6/docs/api/java/lang/Exception.html

C.2. Interception d'une exception

Lorsqu'une **exception** est lancée par une méthode, le programme appelant est généralement interrompu tout de suite. Or ceci n'est pas toujours souhaitable. En effet, Java permet d'intercepter les **exceptions** et de les traiter convenablement pour pouvoir continuer ensuite l'exécution de l'application.

Prenons l'exemple du programme suivant qui utilise la **classe** « **Fraction** » :

```
import java.util.Scanner;
public class Launcher  {
     public static void main(String[] args)  {
          Scanner input = new Scanner(System.in);
          System.out.print("Enter the numerator: ");
          int n = input.nextInt();
          System.out.print("Enter the denominator: ");
          int d = input.nextInt();
          Fraction myFraction = new Fraction(n,d);

          try {
               double result = myFraction.getDecimalValue();
               System.out.println(n+" / "+d+" = "+result);
          }
          catch (ArithmeticException ariExc) {
               System.err.println("Error! Division by zero...");
          }    //System.err est la console d'erreurs (-> messages en rouge)
     }
}
```

La **syntaxe générale** pour intercepter une exception est la suivante :

```
     try
     {
          <instructions>
     }
     catch (<exception> <nom>)
     {
          <instructions_traitement_d'erreurs>
     }
```

S'il se produit une erreur lors de l'exécution des **<instructions>**, alors le programme sort du bloc **try** et regarde si l'exception générée doit être traitée ou non. En effet, il peut y avoir un ou plusieurs blocs **catch** avec des exceptions différentes. **<exception>** est le type de l'exception[2] à traiter et **<nom>** est le nom donné à cette exception (uniquement valable dans le bloc respectif).

Toutes les exceptions et erreurs possèdent les méthodes suivantes qui peuvent être employées pour afficher des détails sur l'exception actuelle :

getMessage()	retourne un message détaillé décrivant l'exception
toString()	retourne un court message décrivant l'exception
printStackTrace()	affiche sur la console d'erreur un message décrivant l'exception actuelle ainsi que tous ses antécédents (→ "**stack trace**")

2 http://java.sun.com/javase/6/docs/api/java/lang/RuntimeException.html

C.3. Plusieurs exceptions dans le même bloc

S'il y a plusieurs blocs **catch**, alors Java exécute le premier bloc qui correspond à l'exception. Il est donc nécessaire de placer les blocs avec des exceptions plus spécifiques DEVANT les blocs plus généraux.

Exemple :

```
try {
    ...
} catch (FileNotFoundException e) {
    System.err.println("FileNotFoundException: " + e.getMessage());
    throw new SampleException(e);

} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

L'exception **FileNotFoundException** est dérivée directement de **IOException**. Si une exception du type **FileNotFoundException** a eu lieu, alors le premier bloc est exécuté. Toutes les autres erreurs du type **IOException** sont traitées par le deuxième bloc. (Remarquez dans l'exemple que la première instruction **catch** n'affiche pas seulement un message, mais lance aussi une autre exception).

Mais attention : Le bloc **catch (FileNotFoundException e)** doit précéder le bloc **catch (IOException e)**, sinon le bloc **FileNotFoundException** ne sera pas pris en compte.

C.4. Le bloc finally

A la fin d'un bloc **try** se trouve très souvent un bloc **finally** qui sera exécuté dans tous les cas, donc même après le traitement d'un bloc **catch**. De cette façon on peut garantir que certaines opérations sont effectuées, même si une erreur imprévue se produit (p.ex. pour fermer un fichier, libérer une imprimante ou une autre ressource, ...). Ainsi la structure générale s'élargit comme suit :

```
try {
     <instructions>
}
catch (<exception1> <nom_1>) {
     <instructions_traitement_d'erreurs_1>
}
catch (<exception2> <nom_2>) {
     <instructions_traitement_d'erreurs_2>
}
catch
     ...
finally {
     <instructions_terminales>
}
```

- Un bloc **try** peut avoir un bloc **finally** même s'il n'y a pas de bloc **catch**.
- Si un bloc **catch** contient une instruction **return**, le bloc **finally** n'est pas exécuté.

En général c'est une bonne idée de placer le code terminal d'une opération ("**cleanup code**") dans un bloc **finally**.

C.5. Le principe "catch-or-throw" & l'instruction "throws"

En Java une exception, une fois lancée, doit être traitée sur place (par un bloc **try-catch**) ou son traitement doit être relégué (lancé, *thrown*) à la méthode appelante. Ce procédé est connu sous le nom "**catch-or-throw**".

Concrètement, si on ne veut pas traiter une exception sur place on peut laisser de côté le bloc **try-catch**. Dans ce cas, il faut ajouter l'instruction **throws** derrière l'en-tête de la méthode et énumérer toutes les exceptions qui ne sont pas traitées dans cette méthode.

La méthode appelante (qui reçoit l'exception via '**throws**') doit elle aussi suivre le principe "catch-or-throw", c.-à-d. elle a aussi la possibilité d'employer **throws** au lieu de **try-catch.** Ainsi une exception peut être passée plusieurs fois à des niveaux d'appels supérieurs, mais au plus tard dans le programme principal, elle doit être traitée par un **try-catch**.

Le compilateur Java veille à ce que le principe "catch-or-throw" soit toujours vérifié.

Exemple :

```
public void printAllDecimalValues(Fraction[] fractions)
          throws ArithmeticException
{
     for(int i=0 ; i<fractions.length ; i++)
          System.out.println(fractions[i].getDecimalValue());
}
```

Instruction pouvant produire une "Arithmetic Exception"

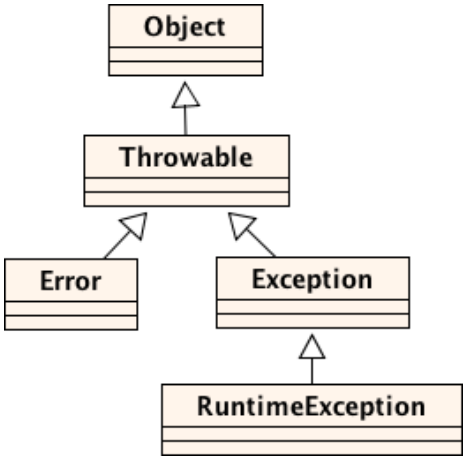
La méthode qui appelle **printDecimalValues** doit aussi suivre le principe "catch-or-throw", c.-à-d. elle doit entourer l'appel d'un bloc **try-catch** (ou bien passer l'exception de son côté à la méthode appelante à l'aide d'une instruction **throws**).

```
...
try
{
     printAllDecimalValues(fractions);
}
catch(ArithmeticException e)
{
     ... ;
}
...
```

Traitement dans la méthode appelante de l'exception reléguée par 'throws'

C.6. Les classes RuntimeException et Error

Voyons d'abord la hiérarchie des classes liées aux erreurs et exceptions (il existe encore quelques douzaines de classes dérivées de **`Error`**, **`Exception`** et **`RunTimeException`** qui ne sont pas représentées ici → voir *JavaDoc* si nécessaire) :



Dans le chapitre précédent, nous avons dit que les exceptions (classe **`Exception`**) doivent suivre le principe "**catch-or-throw**". Il y a deux exceptions au principe "**catch-or-throw**" : les classes **`RuntimeException`** et **`Error`** ainsi que leurs héritiers.

La classe **`RuntimeException`** est la classe ancêtre de toutes les exceptions d'exécution. Les exceptions de ce type peuvent être traitées, mais pas obligatoirement (→ "**unchecked exception**"). De cette façon, le programmeur peut limiter les efforts du traitement d'erreurs et décider lui même si une telle exception doit être traitée ou non. Le plus souvent ces exceptions qui sont dues à des erreurs de programmation et il vaut mieux éliminer l'erreur que de les traiter par un bloc **`try...`**

Toutes les exceptions qui ne sont pas dérivées de **`RuntimeException`** doivent nécessairement être traitées dans la méthode appelante (→ "**checked exception**").

La classe **`Error`** est la classe 'ancêtre' de toutes les erreurs d'exécution. Celles-ci sont dues en principe à des causes externes que le programme ne peut pas prévoir ni y remédier (p.ex. un fichier non accessible à cause d'un problème de hardware → **java.io.IOError**). Une application peut intercepter l'erreur et en informer l'utilisateur, mais il peut aussi être nécessaire d'afficher la pile des erreurs (**`e.printStackTrace();`**) et de terminer l'application.

C.7. Exceptions dans le constructeur

Les exceptions peuvent être utiles dans les constructeurs pour éviter la création d'instances avec des valeurs incorrectes pour les attributs. Jusqu'ici nous n'avons pas encore vu de méthode propre et élégante pour réagir à des paramètres avec des valeurs incorrectes ou inconsistantes lors de la construction d'un objet. Pour cette raison nous n'avons pas défini de test de validité dans les constructeurs ou les manipulateurs. Un utilisateur qui ne respectait pas le domaine de définition d'une classe risquait donc d'obtenir des résultats erronés, sans en être informé.

Une exception permet de sortir du constructeur **sans créer d'instance**. Il est cependant conseillé de tenir compte des réflexions suivantes :

- Éviter dans le constructeur d'affecter l'instance à une variable finale ou à intégrer l'instance dans une collection (p.ex. **`Arraylist`**) avant le test de validité. Sinon, le ramasse-miettes (=*garbage collector*) ne va pas éliminer l'instance incorrecte.

- Produire de préférence une exception traitée (*checked exception*), pour que la méthode appelante puisse réagir à la non-construction de l'objet. Sinon dans la classe appelante, la référence reste **`null`** ou attachée à un autre objet sans que l'utilisateur ne le sache.

C.8. Java 7

- Depuis Java 7 il existe la possibilité de traiter plusieurs exceptions dans un même bloc. Ainsi nous pouvons éviter de devoir définir plusieurs fois le même code dans différents blocs **`catch`**. Pour ce faire, il faut simplement ajouter le symbole '**|**' (***or***) entre les types des exceptions à traiter.

  Exemple :

```
try
    ...
catch (IOException|SQLException ex) {
    ...
}
```

- Depuis Java 7 il existe la possibilité de définir un bloc nommé "**try** *with resources*" pour la création de ressources (fichiers, imprimantes, etc). De cette façon, les ressources ouvertes sont fermées automatiquement sans devoir définir explicitement un bloc **finally**.

  Détails : voir chapitre D.2.try with resources

C.9. Développer sa propre exception

S'il n'existe pas d'exception prédéfinie pour une erreur, nous pouvons la programmer nous même. Supposons par exemple qu'il n'existe pas d'exception pour une division par zéro, alors, nous pouvons la définir de la façon suivante :

```
public class DivisionByZeroException extends ArithmeticException
{
}
```

En fait, cette exception est un simple **héritier** de la classe « **ArithmeticException** », c'est pourquoi son corps est vide. Actuellement nous n'avons pas besoin de fonctionnalités supplémentaires par rapport à celles de la classe « **ArithmeticException** ».

Mais attention : Comme en Java les constructeurs ne sont pas hérités d'une classe à une sous-classe, il n'existe par défaut pas de constructeur où on peut indiquer le message à afficher. Pour ce faire, il faut définir un constructeur avec paramètre et faire appel à **super(...)** . En général, nous redéfinissons les deux constructeurs les plus fréquemment utilisés, par exemple :

```
public class DivisionByZeroException extends ArithmeticException
{
     public DivisionByZeroException(String msg)
     {
          super(msg);
     }

     public DivisionByZeroException()
     {
          super("Division by zero");
     }
}
```

Ensuite, nous pouvons modifier notre classe **Fraction** de la manière suivante :

```
public class Fraction
{
     protected int numerator;
     protected int denominator;

     public Fraction (int numerator, int denominator)
     {
          this.numerator   = numerator;
          this.denominator = denominator;
     }

     public double getDecimalValue()
     {
          if (denominator!=0)
               return (double)numerator/denominator;
          else
               throw new DivisionByZeroException();
     }
}
```

Voici un programme qui utilise les deux classes **DivisionByZeroException** et **Fraction** comme définies ci-dessus:

```
import java.util.Scanner;

public class Launcher
{
     public static void main(String[] args)
     {
          Scanner input = new Scanner(System.in);

          System.out.print("Enter the numerator: ");
          int n = input.nextInt();
          System.out.print("Enter the denominator: ");
          int d = input.nextInt();

          Fraction myFraction = new Fraction(n,d);

          try
          {
               double result = myFraction.getDecimalValue();
               System.out.println(n+" / "+d+" = "+result);
          }
          catch (DivisionByZeroException e)
          {
               System.out.println("Error! Division by zero...");
          }
          catch (Exception e)
          {
               // Erreur inconnue ...
               System.out.println( e.getMessage() );
          }

     }
}
```

- Le premier bloc **catch**, qui traite le cas d'une division par zéro, intercepte une exception du type **DivisionByZeroException**.
- Le deuxième bloc **catch** intercepte une exception générique. Étant donné que toute exception **hérite** de l'exception **Exception**[3], ce bloc intercepte toutes les autres exceptions qui puissent surgir.
- L'instruction **e.getMessage()** retourne via un message textuel le détail sur l'erreur qui s'est produite.
- Remarquez que l'instruction **e.getMessage()** profite pleinement du polymorphisme et de la redéfinition des méthodes avec *'late binding'* (→ voir cours de 2e), c.-à-d. le message affiché ne sera pas le message insignifiant de la classe générique **Exception**, mais celui du type d'exception qui a effectivement été provoquée.

→ voir aussi : http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

3 http://java.sun.com/javase/6/docs/api/java/lang/Exception.html

D. Fichiers

Pour pouvoir mémoriser les données de nos programmes à long terme, il nous manque encore la possibilité de lire et d'écrire dans des fichiers. Il existe deux principes pour la sauvegarde de fichiers :

1. mémorisation des données sous forme de **texte** (p.ex. ligne par ligne ; sous forme de code XML).
   Avantage : nous pouvons visualiser facilement le contenu des fichiers ; les fichiers restent compatibles avec de futures versions du programme ;
   Désavantage : lecture et écriture plus complexe (<=> conversion des données, extraction des données d'une ligne de texte)
2. mémorisation des données sous forme **binaire**, c.-à-d. les données sont sauvegardées exactement comme elles se trouvent dans la mémoire.
   Avantage : lecture et écriture facile des données ;
   Désavantage : nous ne pouvons pas visualiser le contenu du fichier ; chaque modification de la structure des données entraîne une incompatibilité des fichiers avec les versions antérieures. Il faut donc penser à programmer un convertisseur de fichiers pour chaque changement des données.

Commençons avec les aspects communs aux deux solutions :

- La majorité des classes traitées dans ce chapitre se trouve dans le paquet **`java.io`**.
- Un fichier est toujours considéré comme un **flux** de données **(*Stream*).** Un flux peut être un flux d'octets, un flux de caractères, un flux de données (primitives), ou même un flux d'objets.[4]
- Le traitement d'un fichier se passe en trois étapes :
  1. **Ouverture du fichier,**
  2. **Opérations de lecture ou opérations d'écriture,**
  3. **Fermeture du fichier.**
- Les instructions de traitement de fichiers se trouvent en général dans un bloc **try**.
- Les instructions de fermeture d'un fichier se trouvent toujours dans un bloc **finally**.
- Dans ce cours, nous allons traiter uniquement des fichiers à lecture/écriture **séquentielle**. Nous n'allons pas aborder le sujet des fichiers à accès direct. Pour cela, référez-vous à *JavaDoc* ou au tutoriel java (→ *Random Access Files*).
- Lors de l'écriture d'un fichier, le fichier existant du même nom est remplacé par le nouveau fichier. L'ancien fichier du même nom est donc perdu.
- Le chemin (EN:*path*) par défaut est le dossier du projet Java.
- Lors de l'indication d'un nom de fichier, on peut indiquer un chemin complet. Sur tous les systèmes (même en Windows) il faut utiliser le symbole '/' lors de l'indication du chemin.

4 Les flux interviennent aussi dans d'autres opérations en Java, p.ex. dans des opérations sur le réseau, sur les données audio, etc.

D.1. Les fichiers binaires de données simples

A la base des opérations de fichiers binaires en Java se trouvent la lecture et l'écriture de **flux d'octets** (individuels). Ces opérations sont réalisées dans les classes **`java.io.FileInputStream`** et **`java.io.FileOutputStream`**. En pratique, on utilise rarement ces classes de façon directe, car elle ne réalisent que des opérations très élémentaires sur les fichiers.

D'abord, les octets sont lus et écrits l'un après l'autre et chaque opération effectue un nouvel appel au système d'exploitation. Ceci peut devenir extrêmement inefficace : imaginez p.ex. que vous copiez un fichier octet par octet et que pour chaque opération le système doit déplacer la tête de lecture/écriture d'un disque et effectuer une rotation du disque...

Pour cette raison, on emploie en général une mémoire tampon (EN:**Buffer**) pour pouvoir traiter les données par blocs.

Pour accéder à un fichier d'octets de façon convenable, on passe donc le flux par un **flux d'octets avec mémoire tampon** : **`BufferedInputStream`** ou **`BufferedOutputStream`**. Voici les instructions qu'il faut utiliser pour ouvrir un tel fichier en lecture ou en écriture :

```
BufferedInputStream in =
               new BufferedInputStream( new FileInputStream("monfichier.dat") );
BufferedOutputStream out =
             new BufferedOutputStream( new FileOutputStream("monfichier.dat") );
```

Avec un tel fichier, on peut lire/écrire des octets, mais sans savoir de quel type de données il s'agit et sans pouvoir les convertir simplement dans les types de nos applications (int, double, String, etc.).

Pour cela, nous passons le flux d'octets à une classe **`DataInputStream`** ou **`DataOutputStream`** qui sait **lire et écrire des données de types primitifs ainsi que des textes** dans un flux. Nous obtenons les déclarations suivantes pour l'ouverture d'un tel **flux de données primitives** :

```
DataInputStream in = new DataInputStream(
               new BufferedInputStream( new FileInputStream("monfichier.dat")));
DataOutputStream out = new DataOutputStream(
             new BufferedOutputStream( new FileOutputStream("monfichier.dat")));
```

Les méthodes suivantes servent à lire et écrire des données dans un tel fichier :

Type	Lecture	Ecriture
int	**`readInt()`**	**`writeInt()`**
double	**`readDouble()`**	**`writeDouble()`**
float	**`readFloat()`**	**`writeFloat()`**
long	**`readLong()`**	**`writeLong()`**
byte	**`readByte()`**	**`writeByte()`**
boolean	**`readBoolean()`**	**`writeBoolean()`**
char	**`readChar()`**	**`writeChar()`**
String [5]	**`readUTF()`**	**`writeUTF()`**

L'instruction pour fermer le fichier s'appelle simplement **`close()`**.

5 Les textes sont codés en Unicode dans un codage UTF-8 modifié à 1, 2 ou 3 octets par caractère. Chaque texte est précédé automatiquement par la longueur de la chaîne de caractères (long max : 65535 octets).

Lors de la lecture, une exception du type **EOFException** est lancée dès que la fin du fichier est détectée. Ce n'est pas la méthode la plus élégante pour finir la lecture d'un fichier, mais les méthodes **read...** ne savent pas retourner d'autres valeurs (p.ex. **null**) pour annoncer la fin du fichier.

Exemple :

Une séquence de lecture d'un fichier contenant des enregistrements composés d'un texte, d'un entier, et d'un réel peut se présenter comme suit :

```
public void loadFromBinaryFile(String fileName)
                                        throws FileNotFoundException,IOException
{  DataInputStream in = null; // not initialized yet
   try {
      in = new DataInputStream(
                      new BufferedInputStream( new FileInputStream(fileName)));
      System.out.println("You have ordered : ");
      while (true) {
           String description = in.readUTF();
           int    unit        = in.readInt();
           double price       = in.readDouble();
           System.out.println(unit+" units of "+description+" at "+price+"€");
      }
   }
   catch (EOFException e) { //EOF found : normally exiting loop
   }
   finally {
      if (in != null) //in is null if file has not been opened (e.g. wrong filename)
          in.close(); //that's why 'in' has to be known outside the try block
   }
}
```

Dans l'exemple, on a laissé le traitement des exceptions dues au traitement des fichiers à la méthode appelante, qui sait mieux réagir aux problèmes dus p.ex. à un fichier introuvable.

Voici une autre possibilité qui évite la tournure **while(true)** qui n'est pas très élégante :

```
public void loadFromBinaryFile(String fileName)
                                        throws FileNotFoundException,IOException
{  DataInputStream in = null;
   try {
      in = new DataInputStream(
                      new BufferedInputStream( new FileInputStream(fileName)));
      System.out.println("You have ordered : ");
      boolean eof = false;
      while (!eof) {
         try {
            String description = in.readUTF();
            int    unit        = in.readInt();
            double price       = in.readDouble();
            System.out.println(unit+" units of "+description+" at "+price+"€");
         } catch (EOFException e) { //EOF found : normally exiting loop
            eof=true;
         }
      }
   }
   finally {
      if (in != null) //in is null if file has not been opened (e.g. wrong filename)
         in.close(); //that's why 'in' has to be known outside the try block
   }
}
```

Observez que **`try...catch(EOFException e)...`** se trouve maintenant à l'intérieur de la boucle **`while`**.

D.2. *try* with resources

Depuis Java 7 il existe la possibilité de définir un bloc nommé "**try** *with resources*" pour la création de ressources (fichiers, imprimantes, etc). Dans ce cas, on écrit l'instruction pour la création de la ressource (ou des ressources) entre parenthèses immédiatement derrière le mot clé **`try`** et devant l'accolade du bloc. Si la ressource implémente l'interface **`AutoCloseable`**, alors on n'a pas besoin d'ajouter un bloc **`finally`**. La ressource sera fermée automatiquement après le bloc **`try … (catch …)`** même sans définition de **`finally`**.

Comme tous les types de fichiers que nous utilisons dans ce cours définissent *AutoCloseable*, nous pouvons employer *try with resources* dans tous nos exercices, ce qui simplifie sensiblement la notation.

Exemple :

Le bloc suivant (Java 6) :

```
DataInputStream in = null;
try {
      in = new DataInputStream(
               new BufferedInputStream( new FileInputStream(fileName)));
        ...
}
finally {
    if (in != null)
        in.close();
}
```

peut être remplacé par le bloc utilisant '*try with resources*' (Java 7) :

```
try (DataInputStream in = new DataInputStream(
            new BufferedInputStream( new FileInputStream(fileName))) {
    ...
}
```

L'exemple de la page précédente peut être simplifié comme suit :

```
public void loadFromBinaryFile(String fileName)
                                        throws FileNotFoundException,IOException
{  try (DataInputStream in = new DataInputStream(
                    new BufferedInputStream( new FileInputStream(fileName)))) {
      System.out.println("You have ordered : ");
      boolean eof = false;
      while (!eof) {
         try {
            String description = in.readUTF();
            int    unit        = in.readInt();
            double price       = in.readDouble();
            System.out.println(unit+" units of "+description+" at "+price+"€");
         }
         catch (EOFException e) { //EOF found : normally exiting loop
           eof=true;
         }
      }
   }
}
```

D.3. Les fichiers textes

A la base des opérations de fichiers textes en Java se trouvent la lecture et l'écriture de **flux de caractères** (individuels). Ces opérations sont réalisées dans les classes **`java.io.FileReader`** et **`java.io.FileWriter`**. En pratique, on utilise rarement ces classes de façon directe, car elle ne réalisent que des opérations très élémentaires.

Pour accéder à un fichier texte, on emploie des flux de caractères qui sont passés par un flux à mémoire tampon : **`BufferedReader`** ou **`PrintWriter`**[6]. Voici les instructions qu'il faut utiliser pour ouvrir un fichier texte avec mémoire tampon :

```
BufferedReader in  = new BufferedReader( new FileReader("monfichier.txt") );
PrintWriter    out = new PrintWriter   ( new FileWriter("monfichier.txt") );
```

La méthode pour fermer un fichier texte s'appelle **`close()`**.

D.3.1. Lecture dans un fichier texte

La lecture dans un fichier texte passé par un **`BufferedReader`** peut s'effectuer le plus simplement avec les méthodes suivantes :

`int read()` lecture d'un seul caractère. Résultat **`-1`** si le fichier est arrivé à sa fin.

`String readLine()` lecture d'une ligne de texte (les symboles de fin de ligne ne sont pas retournés comme résultat). Résultat **null** si le fichier est arrivé à sa fin.

Lors de la lecture d'un fichier texte, nous n'avons donc pas besoin de provoquer une exception pour reconnaître la fin du fichier, mais nous pouvons simplement contrôler le résultat des méthodes **`read`** ou **`readLine`**.

S'il y a plusieurs données mémorisées dans chaque ligne de texte, alors vous devez les extraire vous même.

Remarque :

Comme alternative à un **`BufferedReader`**, il peut être intéressant d'employer un **`Scanner`** pour la lecture. Ainsi, vous pouvez employer les méthodes pratiques définies dans la classe **`java.util.Scanner`** : **`hasNext()`**, **`next()`**, **`nextInt()`**, **`nextDouble()`**, etc. (→ voir *JavaDoc* ou *Annexe* du cours de 3e).

D.3.2. Ecriture dans un fichier texte

La classe **`PrintWriter`** possède les méthodes **`print(...)`** et **`println(...)`** que vous connaissez de **`System.out`**. En fait, **`System.out`** n'est rien d'autre qu'un **`PrintWriter`**. De cette façon, il est possible d'écrire toutes sortes de données dans un fichier texte.

D.3.3. Retours à la ligne

Un avantage de l'utilisation des méthodes **`BufferedReader.readLine`** et **`PrintWriter.println`** est que ces méthodes reconnaissent automatiquement les différentes séquences de fin de ligne possibles ('**\n**' line-feed ; '**\r**' carriage-return ; "**\r\n**" carriage-return & line-feed). **`println`** insère le symbole de fin de ligne du système d'exploitation actuel.

6 Il existe aussi une classe **BufferedWriter**, mais les méthodes offertes par **BufferedWriter** sont tellement limitées, qu'il vaut mieux employer **PrintWriter**. **PrintWriter** passe aussi par une mémoire tampon.

Voici une classe qui sait copier un fichier ligne par ligne en remplaçant les symboles de fin de ligne par les symboles du système d'exploitation actuel.

En profitant du fait que la structure *try with resources* peut même intégrer l'ouverture et la fermeture de plusieurs fichiers, à partir de Java 7 nous pouvons employer la version simplifiée :

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
   public static void main(String[] args) throws IOException {
     try (
      BufferedReader in = new BufferedReader(new FileReader("mytext.txt"));
      PrintWriter    out= new PrintWriter(new FileWriter("mytext_copy.txt"))) {
           String l;
           while ((l = in.readLine()) != null) {
                out.println(l);
           }
      }
   }
}
```

D.3.4. Lecture et sauvegarde de couleurs

Une couleur peut être sauvegardée et lue dans un fichier texte en la convertissant dans un entier, puis dans un texte avant de la sauvegarder :

sauvegarde : conversion en un entier :

```
out.print( color.getRGB() );
```

lecture : reconversion en une couleur :

```
color = Color.decode(texteLu);
```

ou aussi :

```
color = new Color(Integer.valueOf(texteLu));
```

D.4. Les fichiers binaires d'objets

Java nous offre une possibilité très simple pour mémoriser des objets avec tout leur contenu dans un fichier. Ceci se fait par des **flux d'objets** : **`ObjectInputStream`** et **`ObjectOutputStream`**.

La condition pour pouvoir sauvegarder les objets est qu'il soient '**sérialisables**'. Ceci veut dire qu'ils implémentent l'interface '**Serializable**' (→ voir chapitre E.Les interfaces). La plupart des classes prédéfinies en Java implémentent déjà cette interface. Ainsi, il suffit le plus souvent que nous ajoutions '**`implements Serializable`**' derrière la déclaration de nos classes que nous voulons sauvegarder.

Ensuite, nous pouvons employer les méthodes **`writeObject`** et **`readObject`** pour écrire ou lire un objet dans un fichier.

Malheureusement, **`readObject`** ne retourne pas **`null`** à la fin du fichier, mais provoque une exception du type **`EOFException`**. Pour lire une série d'objets, on peut donc procéder comme pour la lecture d'un fichier de données binaires, ou mieux : on peut mémoriser le nombre d'objets en premier lieu dans le fichier (p.ex. avec **`writeInt`**).

Voici deux méthodes qui servent à lire et à écrire toute une liste d'objets dans un fichier :

```
public void saveToFile(String fileName) throws IOException
{
   try ( ObjectOutputStream obj_out =
               new ObjectOutputStream(new FileOutputStream(fileName)) ) {
      obj_out.writeObject(list);
   }
}


public void loadFromFile(String fileName) throws IOException
{
   try ( ObjectInputStream obj_in =
                new ObjectInputStream(new FileInputStream(fileName)) ) {
      list = (ArrayList)obj_in.readObject();
   }
   catch(ClassNotFoundException e) {
      System.out.println("Error reading Object File: "+e);
   }
}
```

Attention :

Dès qu'une donnée change dans une des classes à sauvegarder, il est possible que les fichiers ne soient plus compatibles avec les versions des fichiers sauvegardées antérieurement. Ceci est un problème qui n'est pas si facile à prévoir ni à résoudre. Pour cette raison, il est indiqué de ne pas employer la sérialisation à la légère et de consulter en détail les documents de *JavaDoc* et des tutoriels de Java avant d'employer la sérialisation dans des projets professionnels.

D.5. Ajout à la fin d'un fichier ("Append")

Pour ajouter des données à la fin d'un fichier texte ou binaire existant, il suffit d'ouvrir le fichier en envoyant ***true*** comme deuxième paramètre (le paramètre 'append'). Si le fichier n'existe pas encore, il est créé automatiquement.

Exemples :

Fichier texte :

```
PrintWriter out = new PrintWriter(new FileWriter("monfichier.txt", true));
```

Fichier binaire de données :

```
DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream( new FileOutputStream("monfichier.dat", true)));
```

Remarque pour avancés (ne fait pas partie du cours) :

Pour un fichier binaire d'objets, la situation est différente, puisque le début du fichier contient une en-tête (***header***) qui décrit de la structure des objets mémorisés. Cette en-tête ne doit pas être répétée lors de l'ajout d'objets.

En conséquence :

1. La première fois, le fichier doit être écrit normalement, pour que l'en-tête soit bien sauvegardée.
2. Lors des ajouts, il faut éviter l'écriture de l'en-tête en redéfinissant la méthode ***writeStreamHeader***.

En plus, il faut écrire les objets individuellement, c.-à-d. on ne peut pas sauvegarder une liste entière d'objets en une seule instruction, sinon, le fichier contiendra une série de listes d'objets, mais lors de la lecture, on n'obtiendra que la première liste qui se trouve dans le fichier.

```
public void saveToObjectFile(String fileName) throws IOException {
   if (!(new File(fileName).exists())) { //new file => write WITH header
      try (ObjectOutputStream out = new ObjectOutputStream
                                              (new FileOutputStream(fileName))) {
          for (int i = 0; i < fractionList.size(); i++) {
              out.writeObject(fractionList.get(i));
          }
      }
   } else {//file exists => write WITHOUT header
      try (ObjectOutputStream out =
                new ObjectOutputStream(new FileOutputStream(fileName, true)) {
          @Override
          protected void writeStreamHeader() throws IOException {
               reset(); //disregard the state of objects already written
          } };) {
          for (int i = 0; i < fractionList.size(); i++) {
             out.writeObject(fractionList.get(i));
          }
      }
   }
}
```

D.6. Les fichiers XML

Un fichier XML (*Extensible Markup Language)* n'est rien d'autre qu'un simple fichier texte qui suit des règles bien définies[7]. Il s'en suit que l'écriture d'un fichier XML n'est rien d'autre que l'écriture d'une chaîne de caractères suivant ces règles, qui sont – en résumé – les suivantes :

- La première ligne indique la version XML utilisée ainsi que l'encodage du fichier.
- Chaque fichier XML possède un élément de base unique. En anglais on parle du « root element ».
- Les noms des balises (EN: *Tags*) peuvent être librement choisis mais doivent être conformes aux normes communément connues. Les noms des balises et attributs sont généralement écrits en minuscules, respectivement suivent la notation Camel[8] telle qu'elle est aussi connue en Java.
- Toute balise qui est ouverte doit obligatoirement être fermée !
- Tout comme en HTML, certains caractères tels que le « & » ou « < », doivent être encodés :
  - & → &
  - < → <
  - …
- Un fichier XML peut suivre une définition de document type[9]. On parle de « DTD » *(Document Type Definition)*. Or ceci n'est pas nécessaire ni traité dans le présent cours.
- On peut aussi rattacher une feuille de style XSL (*Extensible Stylesheet Language)*[10] à un fichier XML. Ceci n'est pas non plus traité dans le présent cours, mais il est bon de savoir que cela existe !

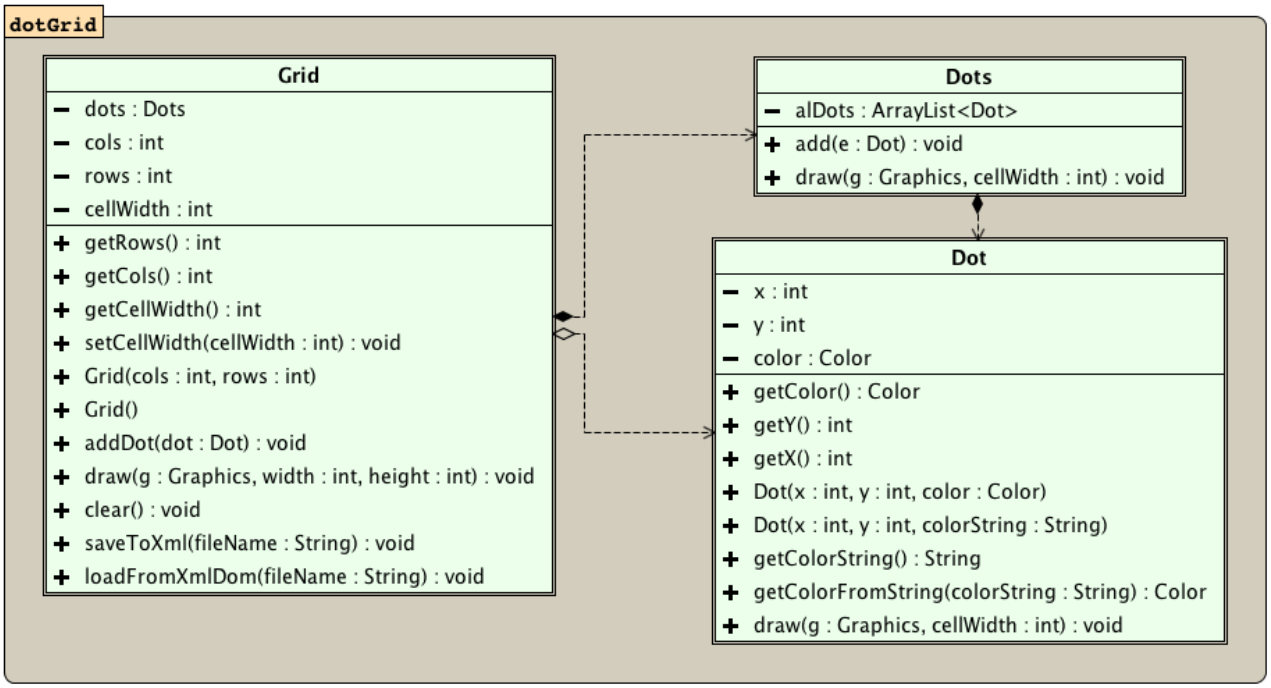
Exemple :

```
<?xml version="1.0" encoding="UTF-8" ?>
<dots>
  <alDots>
    <dot>
      <x>1</x>
      <y>1</y>
      <color>
        <red>255</red>
        <green>255</green>
        <blue>0</blue>
        <alpha>255</alpha>
      </color>
    </dot>
    <dot>
      <x>2</x>
      <y>1</y>
      <color>
```

7 http://en.wikipedia.org/wiki/XML
8 http://en.wikipedia.org/wiki/CamelCase
9 http://en.wikipedia.org/wiki/Document_Type_Definition
10 http://en.wikipedia.org/wiki/XSL

```
        <red>255</red>
        <green>0</green>
        <blue>0</blue>
        <alpha>255</alpha>
      </color>
    </dot>
...
  </alDots>
</dots>
```

Le fichier XML ci-dessus représente une grille avec des points colorés à certaines positions. Le diagramme de classe UML correspondant pourrait être le suivant :



D.6.1. Les analyseurs syntactiques

Afin de lire un fichier XML, il existe deux méthodes :

- L'analyseur syntaxique (EN : *parser*) du type « DOM » *(Document Object Model)* copie la structure entière d'un fichier XML dans la mémoire sous forme d'un grand arbre. L'avantage est qu'on aura à tout moment accès à toutes les informations. Elle convient pour de petits fichiers, tandis que la consommation de mémoire sera trop importante pour des fichiers très grands.
- L'analyseur syntaxique « SAX » *(Simple API for XML)* n'a pas besoin de copier en mémoire le fichier entier. Il parcourt le fichier séquentiellement et jette des événements lors de la rencontre des différents éléments du fichier. Le programme peut réagir ou non aux différents événements lancés. Ceci a l'avantage d'être utilisable même pour de gigantesques fichiers. Le désavantage est que l'entièreté des informations n'est pas accessible à tout moment.

Pour simplifier, nous allons seulement employer l'analyseur **Dom** dans ce cours.

D.6.2. La classe XStream

Pour simplifier la lecture et l'écriture de fichiers, nous allons faire recours à la bibliothèque **XStream** qui a l'avantage d'effectuer automatiquement l'analyse des objets à sauvegarder et elle sait rétablir les objets à partir du fichier et de la définition des classes. Ceci est possible grâce au **'reflection engine'** (→ ***java.lang.reflect***) de Java, qui permet à un programme Java d'accéder à la définition des classes et de tous leurs composants.

Utilisation de la bibliothèque :

- Téléchargez d'abord *XStream* du site http://x-stream.github.io
- Créez un dossier **lib** dans le dossier du projet où vous voulez utiliser des fichiers XML. Copiez-y le fichier **jar** principal de *XStream* (p.ex. : **xstream-1.4.8.jar** , le nom change avec la version)
- Ouvrez le projet en NetBeans et ajoutez le fichier **jar** à votre projet : clic droit sur '**Libraries**', puis sélectionnez 'Add JAR/folder...'. Veillez à faire une référence relative au fichier.
- Dans la classe qui contiendra les opérations de lecture et d'écriture, importez la classe XStream et celle du gestionnaire *Dom* : *com.thoughtworks.xstream.XStream*, *com.thoughtworks.xstream.io.xml.DomDriver*

D.6.2.1. Ecriture du fichier :

Une version de base pour écrire le fichier avec les points colorés peut se présenter comme suit :

```
public void saveToXml(String fileName) throws IOException {
    XStream xstream = new XStream(new DomDriver("UTF-8"));
    String xml = xstream.toXML(dots);

    try (PrintWriter out =  new PrintWriter(new FileWriter(fileName))) {
           out.println("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>");
           out.println(xml);
    }
}
```

Réalisez ce programme et essayez de rapprocher la représentation xml de celle montrée plus haut en employant les instructions ***xstream.setMode****(...)* *et* ***xstream.alias****(...)*

D.6.2.2. Lecture du fichier :

Une version de base pour lire le fichier avec les points colorés peut se présenter comme suit :

```
public void loadFromXmlDom(String fileName) throws IOException {
    XStream xstream = new XStream(new DomDriver("UTF-8"));
    try (BufferedReader in = new BufferedReader(new FileReader(fileName))){

             dots = (Dots)xstream.fromXML(in);

    } catch (com.thoughtworks.xstream.mapper.CannotResolveClassException e) {
        JOptionPane.showMessageDialog(null,
               "Cannot resolve XML Tags in file '"+fileName+"'",
                "XML read Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

[Remarque: XStream permet aussi de traiter des fichiers JSON:
voir http://x-stream.github.io/json-tutorial.html]

D.6.2.3. Remarque pour les versions XStream > 1.4.17 :

Pour éviter que des fichiers XML puissent être utilisés pour introduire du code malévolent dans un programme ou un système, les version actuelles de *XStream* nous forcent de déclarer explicitement les classes que nous avons prévues dans notre fichier XML.
Lors de la lecture d'un fichier XML, nous devons employer les méthodes **allowTypes** ou **allowTypesByWildcard** de la classe *XStream* pour déclarer les classes permises à la lecture.

Exemples :

```
xstream.allowTypes(new Class[] { Dot.class, Dots.class });
xstream.allowTypesByWildcard(new String[] {"figures.*"}); //package name +".*"
```

D.6.2.4. Remarque pour JDK 17 :

For compatibility with JDK17, classes saved in xml MUST be accessed from reflection engine.
==> see: https://x-stream.github.io/faq.html#Compatibility

Here it says:

"XStream fails since Java 17, because types in modules cannot be accessed from the unnamed module!

Again, this is normal. The reflection stuff is required to get all required information to recreate an instance of a Java type at unmarshalling time. However, since Java 17 it is no longer possible to allow this access with a single runtime option. You have to open all packages of the individual modules for the unnamed module with the option --add-opens, where XStream requires access,
e.g. `--add-opens java.base/java.util=ALL-UNNAMED`

If for example, instances of *java.awt.Point* and/or *java.awt.Color* are saved
=> add the following line to project's VM Options:

```
--add-opens java.desktop/java.awt=ALL-UNNAMED
```

To make it work in the compiled **jar**, too, add the following line to *build-imp.xml* , below the line:
`<attribute name="Main-Class" value="${main.class}"/>` :

```
<attribute name="Add-Opens" value="java.desktop/java.awt"/>
```

(This line has the "Add-Opens" directive added to **manifest.mf** inside the jar file)

If more than one package has to be accessed, there can be many compiler instructions to be added.

Example :

```
--add-opens java.base/java.lang.reflect=ALL-UNNAMED --add-opens
java.base/java.util=ALL-UNNAMED --add-opens java.base/java.text=ALL-UNNAMED --
add-opens java.desktop/java.awt=ALL-UNNAMED --add-opens
java.desktop/java.awt.font=ALL-UNNAMED
```

D.7. Localisation des fichiers

D.7.1. Dialogues d'ouverture et de sauvegarde

Pour ouvrir ou sauvegarder des fichiers, il est pratique d'employer des dialogues prédéfinis de Java qui permettent de choisir des fichiers sur un support électronique (disque, USB, réseau, etc.). Ces dialogues n'effectuent pas d'action de sauvegarde ou de lecture : elles nous permettent uniquement de choisir un chemin et un nom de fichier de façon confortable.

Pour cela, nous pouvons utiliser la classe **JFileChooser** et ses méthodes :

Constructeur	**JFileChooser(** <*fichier initial*>)
int **showSaveDialog**(<*parent*>)	ouvre un dialogue de sauvegarde. <*parent*> définit la fiche qui possède le dialogue, ce paramètre influence la présentation et la position du dialogue. Le plus souvent, nous utiliserons **this** ou **null** pour ce paramètre. La valeur entière retournée par la méthode peut avoir l'une des valeurs suivantes : JFileChooser.**CANCEL_OPTION** : le dialogue a été fermé avec le bouton '*Cancel*' ou la touche '*escape*'. L'action doit donc être annulée. JFileChooser.**APPROVE_OPTION** : le dialogue a été fermé avec le bouton '*Ok*' ou la touche '*enter*'. L'action doit donc être exécutée. JFileChooser.**ERROR_OPTION** : le dialogue a été fermé avec une erreur. L'action doit donc être annulée.
int **showOpenDialog**(<*parent*>)	ouvre un dialogue d'ouverture de fichier. - voir ***showSaveDialog*** -
File **getSelectedFile**()	retourne le ficher sélectionné dans le dialogue
String **getName**(<*File*>)	retourne le nom (sans le chemin) du fichier donné comme paramètre
addChoosableFileFilter(<*FileFilter*>)	Réduit la liste des fichiers affichés en affichant uniquement les fichiers donnés par le ou les filtres. ***- Définir des filtres : voir ci-dessous -***
setAcceptAllFileFilterUsed(false)	supprimer le filtre *"All Files"* qui permet d'afficher tous les fichiers

Pour obtenir le nom du fichier sélectionné dans un dialogue nommé *fileChooser* **AVEC** le chemin oú se trouve le fichier sur le disque, on peut écrire :

```
String currentFile = fileChooser.getSelectedFile().getAbsolutePath();
```

(-→ voir aussi chap : D.8)

D.7.2. Définition d'un filtre de fichiers :

Il est possible de définir des filtres de fichiers personnels en surchargeant les méthodes **getDescription** et **accept** de **FileFilter**. Depuis Java 6, il existe cependant une méthode beaucoup plus confortable : la classe **FileNameExtensionFilter** permet de définir des filtres de fichiers personnalisés en une seule instruction :

```
FileFilter ffilter = new FileNameExtensionFilter("JPEG file", "jpg", "jpeg");
```

crée un filtre décrit comme *"JPEG File"* qui limite l'affichage aux fichiers portant les extensions *jpg* ou *jpeg* (en majuscules ou en minuscules).

Autres exemples :

```
FileFilter ff = new FileNameExtensionFilter("Images","jpeg","jpg","gif","png");
FileFilter ff = new FileNameExtensionFilter("Text files (*.txt)","txt");
```

Exemple pour l'ouverture d'un fichier :

```
JFileChooser fc = new JFileChooser(currentFile); //currentFile du type File
fc.setAcceptAllFileFilterUsed(false);
fc.addChoosableFileFilter(new FileNameExtensionFilter("Draw files","drw"));

if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
   currentFile = fc.getSelectedFile();
   drawPanel.loadFromFile(currentFile.getAbsolutePath());
}
```

Pour obtenir le dossier de travail du projet actuel (ou du programme actuel), vous pouvez vous référer aux informations du système :

```
System.getProperty("user.dir")
```

Pour obtenir le dossier des documents de l'utilisateur, vous pouvez employer :

```
System.getProperty("user.home")
```

Dans la plupart des projets, c'est une bonne idée d'employer un attribut '**currentFile**' et de l'initialiser avec l'un des deux chemins décrits ci-dessus. Ainsi, l'utilisateur n'a pas besoin de parcourir chaque fois tout le chemin jusqu'aux fichiers qui appartiennent à votre projet.

D.8. Opérations sur fichiers et chemins d'accès

Jusqu'ici, nous avons travaillés avec des fichiers, sans nous occuper de leur existence ou conditions d'accès. La classe **java.io.File** permet de travailler simplement avec des fichiers et des chemins d'accès.

Voici un bref aperçu sur les méthodes les plus usuelles disponibles pour un objet du type **File** :

- Un objet du type **File** peut être construit à partir d'un nom de chemin (avec des séparateurs '\') ou d'un objet **URI** (descripteur universel de ressources).
- Pour obtenir le chemin absolu d'un fichier, on peut employer la méthode **getAbsolutePath** . P.ex :

  ```
  String path = openDialog.getSelectedFile().getAbsolutePath()
  ```

- La méthode **exists** permet de tester si un tel fichier existe déjà.
- Les méthodes **canExecute, canRead, canWrite** permettent de détecter les droits d'accès de l'utilisateur sur un fichier.
- La méthode **delete** permet d'effacer un fichier.
- Les méthodes **isDirectory** et **isFile** permettent de détecter si on a à faire à un fichier ou à un dossier.
- La méthode **length** retourne la taille du fichier en octets ou zéro si le fichier n'existe pas.
- La méthode **mkdir** et **mkdirs** permettent de créer un dossier en créant ou non les sous-dossiers qui manquent.
- Les méthodes **getTotalSpace** et **getFreeSpace** retournent en octets l'espace total et l'espace encore disponible sur le support du fichier.
- La méthode **renameTo** permet de renommer un fichier.

Exemple :

Voici une méthode statique qui retourne **true** si un fichier donné existe :

```
public static boolean fileExists(String fileName) {
      java.io.File f = new java.io.File(fileName);
      return f.exists();
}
```

D.9. Fichiers de ressources dans un projet

Souvent, on a besoin de ressources supplémentaires (fichiers son, texte ou image) pour faire fonctionner un projet. Ces fichiers doivent être disponibles dès que le programme démarre et ceci aussi bien si le projet est en développement que plus tard dans la version compilée (fichier .**jar**).

Exemple pour ajouter un dossier contenant des images à un projet :

Ajouter un dossier **pics** au dossier **src** de votre projet. Ainsi, lors de la compilation, ce dossier sera ajouté automatiquement au dossier '**build/classes**' et dans le dossier racine du fichier '**jar**'. L'accès aux fichiers (images) se fait alors par :

```
try {
    URL myurl = this.getClass().getResource("/pics/Picture1.png");
    image = ImageIO.read(myurl);
} catch (IOException ex) {
    ...
}
```

Exemple pour un fichier texte qui se trouve sous **'.../src/resources/text.csv'** :

```
InputStream inStream = getClass().getResourceAsStream("/resources/text.csv");
if(inStream == null)
   throw new FileNotFoundException();
else
   try (BufferedReader in=new BufferedReader(new InputStreamReader(inStream)))
     {
        ...
     }
```

D.10. Charger au démarrage et sauvegarder à la fin

Parfois il est utile de charger un fichier dès le démarrage d'un programme et de le sauvegarder dès que le programme est fermé.

Exemples :

- Actualiser les meilleurs scores d'un jeu,
- mémoriser et rétablir l'état des options (préférences) de l'utilisateur.

Pour charger des fichiers au démarrage, il suffit d'ajouter les opérations nécessaires au constructeur de la classe principale.

Pour sauvegarder des fichiers lors de la fermeture, le cas est plus difficile, puisqu'il faut détecter l'action de fermeture et la retarder jusqu'à ce que le fichier soit écrit.

Une solution serait de définir l'événement **WindowClosing** de la fiche principale du programme et d'y effectuer toutes les opérations nécessaires.

Attention : L'événement **WindowClosing** est lié à la fiche et n'est pas évalué lorsqu'on ferme la fiche de façon forcée (p.ex. par *System.exit(0), Task Manager, Force quit, ...*).

Une autre possibilité plus sûre, p.ex. pour sauvegarder les meilleurs scores, est de sauvegarder le fichier à chaque changement.

Remarque :

Pour sauvegarder les préférences de l'utilisateur, il existe un paquet spécial **java.util.prefs** qui permet de sauvegarder les préférences à l'endroit prévu par la plateforme sur laquelle tourne le programme. Le programmeur n'a pas besoin de s'occuper des détails de l'implémentation. Il existe deux arborescences pour les préférences : préférences 'utilisateur' et 'système'. La majorité de nos informations devra être sauvegardée dans l'arborescence des préférences 'utilisateur'. Pour les détails, recherchez l'aide et le tutoriel sur *'Preferences API'*.

E. Les interfaces

Dans certains langages de programmation (p.ex C++), une classe peut hériter de plusieurs classes de base. On appelle cela **héritage multiple** (EN : '**multiple inheritance**'). A cause de la complexité d'une telle architecture et des difficultés qui peuvent en provenir (p.ex. l'héritage d'une méthode du même nom de plusieurs classes différentes/héritage en losange), les créateurs de Java ont décidé de ne pas implémenter l'héritage multiple en Java.

On a cependant jugé utile d'implémenter la possibilité que plusieurs classes de structures hiérarchiques complètement différentes puissent hériter les mêmes définitions pour leurs **interfaces**.

Du point de vue de la programmation, une interface est un type spécial d'une classe, qui contient uniquement des **méthodes abstraites**[11] et des **constantes**. La définition d'attributs ou de constructeurs n'est pas permise. Lors de la définition, le mot clé **`class`** est remplacé par **`interface`**. Le fait qu'une classe hérite ou plutôt **réalise une interface** est marqué par le mot-clé **implements** (au lieu de **`extends`**). Une classe peut hériter d'une classe de base et implémenter plusieurs interfaces à la fois. Si une classe implémente une interface, elle peut être traitée comme si elle était une sous-classe de l'interface.

Du point de vue conceptuel, une interface peut être considérée comme un contrat ou une 'promesse' : En employant le mot clé **`implements`**, une classe s'engage à réaliser ou redéfinir toutes les méthodes prescrites par l'interface. Chaque interface a une utilité et des spécificités différentes. Pour connaître les spécifications exactes des interfaces ainsi que les exceptions qu'elles sont supposées lancer, consulter **JavaDoc**.

L'utilité des interfaces devient plus claire en considérant quelques exemples. Les interfaces les plus fréquentes sont :

`Comparable` : permet la comparaison des instances d'une classe à l'aide de **`compareTo(...)`**

`Serializable` : permet la sauvegarde des instances d'une classe dans un fichier (ou d'autres médias persistants)

`Iterable`, **`Iterator`** , **`ListIterator`** : permettent de parcourir une **collection** d'objets (à l'aide de **`hasNext()`**, **`next()`**) sans savoir exactement comment les objets sont mémorisés (voir collections : **`Map`**, **`List`**, **`ArrayList`**, ...). Optionnellement l'itérateur peut définir la méthode **`remove()`** qui sert à supprimer des objets. Les itérateurs sont souvent employés pour parcourir les éléments d'une collection à l'aide de la structure *'**`for`**-each'.*

`Cloneable` : permet la copie exacte des instances d'une classe à l'aide de **`clone()`**. Cas spécial : **`.clone`** est déjà défini dans **`Object`**, mais **`implements Cloneable`** garantit qu'une copie correcte (de tous les champs) est garantie. Une classe implémentant **`Cloneable`** possède généralement une méthode publique **`.clone`** qui redéfinit la méthode **`.clone`** héritée.

Dans la programmation GUI en Swing, on emploie les fameux interfaces **`...Listener`**.

11 *Depuis Java 8, une interface peut aussi contenir des méthodes 'default' contenant le comportement par défaut.*

Remarques :

- Les interfaces peuvent aussi hériter d'une autre interface pour l'étendre, p.ex. pour ajouter une autre méthode à l'interface.
- Il est possible de créer une instance d'une interface. (→ voir code généré par NetBeans...)
- Les interfaces sont souvent définies comme **types paramétrés** (EN : **generics**), en indiquant le nom de le classe entre '<' et '>'. Par exemple : **`implements Iterable<Shape>`** indique que la classe sait fournir un itérateur qui retourne des éléments du type **`Shape`**.

Approfondir les notions sur les interfaces dans ***JavaDoc*** et les documents suivants :

- http://download.oracle.com/javase/tutorial/java/IandI/createinterface.html
- http://www.codestyle.org/java/faq-Interface.shtml

Exemple :

Imaginons que nous ayons un projet avec plusieurs hiérarchies de classes et d'objets. Nous aimerions que tous les objets sachent imprimer leurs informations actuelles à l'écran.

Une première idée pourrait être de définir une classe abstraite **`Printable`** et d'en dériver tous les objets de notre projet. Ceci serait quand même assez artificiel, et détruirait la logique de l'héritage des classes de notre projet. En plus, si nous utilisons dans notre projet des objets dérivés de classes prédéfinies, il ne serait pas possible d'intercaler notre classe **`Printable`** dans la hiérarchie d'héritage.

La solution est alors de définir une interface **`Printable`** contenant la déclaration d'une méthode abstraite **`print()`**. Nous ajoutons à toutes les classes de notre projet l'instruction **`implements Printable`** et nous implémentons dans toutes les classes la méthode **`print()`**. De cette façon, la hiérarchie originale de nos classes n'est pas affectée, mais tous les objets ont la capacité d'imprimer leurs informations à l'écran.

Aspect de l'"Héritage multiple" :

Nous pouvons traiter tous les objets implémentant **`Printable`** comme s'ils héritaient non seulement de leur superclasse, mais aussi de la classe **`Printable`**. Nous pouvons donc parcourir une liste d'objets complètement différents, mais qui implémentent '**`Printable`**' et faire appel à leur méthode **`print()`** - tout comme s'ils héritaient tous de cette classe !

Comme en réalité chaque classe ne possède qu'une seule classe mère, on évite cependant toutes les difficultés émanant d'un vrai héritage multiple.

F. La récursivité

F.1. Définition et discussion

F.1.1. Définition

récursivité: le fait qu'une méthode (plus généralement : un sous-programme) s'appelle elle-même directement ou indirectement

Si une méthode s'appelle elle-même directement, on dit qu'elle est **récursive**.
Si la récursivité concerne un groupe de méthodes, on parle de **récursivité indirecte** ou on dit qu'elles sont **mutuellement récursives**.

F.1.2. Exemple

<table>
<tr><th colspan="2">Calcul de la factorielle d'un nombre naturel</th></tr>
<tr><th>Solution itérative</th><th>Solution récursive</th></tr>
<tr><td>Définition mathématique:

$\begin{cases} 0! = 1 \\ N! = 1 \cdot 2 \cdot \ldots \cdot N \quad (\forall N \in IN^*) \end{cases}$</td><td>Définition mathématique:

$\begin{cases} 0! = 1 \\ N! = N \cdot (N-1)! \quad (\forall N \in IN^*) \end{cases}$</td></tr>
<tr><td>Programmation:


```
public int factorial(int n)
{
     int result = 1;
     for (int i=1 ; i<=n ; i++)
          result = result * i;
     return result;
}
```

Boucle</td><td>Programmation:


```
public int factorial(int n)
{
     if (n==0)
          return 1;
     else
          return n * factorial(n-1);
}
```

Appel récursif</td></tr>
<tr><td>Evaluation pour n=4

4! = 1
↘
... · 2
↘
... · 3
↘
... · 4
= 24</td><td>Evaluation pour n=4

4! = 4 · 3!
= 4 · (3 · 2!)
= 4 · (3 · (2 · 1!))
= 4 · (3 · (2 · (1 · 0!)))
= 4 · (3 · (2 · (1 · 1)))
= 4 · (3 · (2 · 1))
= 4 · (3 · 2)
= 4 · 6
= 24</td></tr>
</table>

F.1.3. La condition de terminaison

Lors du codage d'une méthode récursive, il est impératif d'inclure une condition de terminaison, autrement dit un moyen de mettre un terme au calcul. Sans cette condition de terminaison, le sous-programme continuera à s'appeler indéfiniment, consommant de plus en plus de mémoire et de temps, jusqu'à ce qu'une erreur se produise et que le calcul s'interrompe.

Considérons la méthode **`factorial`**:

- Lorsque **`n`** est égal à 0, la méthode renvoie 1. C'est le seul cas qui n'initie pas d'appel récursif. La condition '**`n==0`**' est donc la condition de terminaison pour la méthode **`factorial`**.
- Pour toutes les valeurs positives de **`n`** est générée une suite d'appels récursifs qui commence par **`factorial(n-1)`**. A chaque appel récursif **`n`** diminue, pour aboutir enfin à l'appel **`factorial(0)`**, c.-à-d. le cas qui remplit la condition de terminaison. Ensuite les résultats de tous les appels récursifs sont calculés 'à rebours' en commençant par le dernier appel récursif. Enfin l'expression **`n*factorial(n-1)`** est évaluée et la valeur est retournée comme résultat de l'appel initial.
- Pour toutes les valeurs négatives de **`n`** est générée une suite d'appels récursifs qui commence par **`factorial(n-1)`**. A chaque appel récursif, **`n`** diminue sans jamais aboutir à un appel remplissant la condition de terminaison. Ainsi est générée une suite 'infinie' d'appels récursifs.

  Remarque: Comme en mathématiques la fonction factorielle est définie uniquement pour des entiers naturels, la réaction de la méthode pour un nombre négatif peut être négligée. En principe, il est sous la responsabilité de l'utilisateur de la méthode de s'assurer que les données correspondent au domaine de définition de la fonction.

Guide pratique:

En conclusion, vous devez vous assurer que les points suivants sont observés lors de la construction d'une méthode récursive:

- les conditions et les valeurs pour les cas de base (c.-à-d. les cas qui ne mènent pas à des appels récursifs) doivent être trouvées,
- l'appel récursif doit être défini de manière conditionnelle: à chaque appel, il doit y avoir une vérification de la/des condition(s) d'arrêt,
- à chaque appel récursif, un ou plusieurs des paramètres qui sont transmis au sous-programme doivent se rapprocher de la condition d'arrêt,
- le nombre d'appels récursifs pour parvenir à un résultat doit être fini (s'il était infini, on aurait une boucle sans fin),
- dans un sous-programme récursif, la complexité du problème doit être réduite à chaque nouvel appel récursif.

"Pour comprendre la récursivité, il faut tout d'abord comprendre la récursivité ..."
"Cette phrase est fausse" (paradoxie d'Epimenides)

F.1.4. Pourquoi utiliser la récursivité?

La question est: pourquoi utiliser la récursivité alors qu'il existe des méthodes non récursives qui sont tout aussi efficaces pour effectuer la même tâche?

- Les méthode récursives ont tendance à être plus compactes et souvent plus élégantes que les solutions itératives.
- Certains problèmes mathématiques sont définis de façon récurrente et se laissent résoudre de façon très élégante à l'aide de la récursivité.
- Certains problèmes sont résolus en appliquant directement la récursivité, et seraient insolvables (ou difficilement solvables) sans récursivité.
- La récursivité est particulièrement adaptée au traitement de structures de données définies récursivement, comme les arbres (traités plus tard dans l'année).

Inconvénients de la récursivité

- La récursivité nécessite un minimum de pratique pour être maîtrisée. Alors que l'itération vient naturellement, ce n'est pas le cas de la récursivité.
- La récursivité est gourmande en ressources. Chaque appel d'une méthode récursive génère un temps système aussi important que tout appel d'une méthode non récursive.
- La pile (*stack*) est utilisée pour stocker l'état de la méthode appelante, de sorte que lorsque la méthode appelée a terminé, le traitement est renvoyé à l'appelant. Une erreur de dépassement de pile se produit lorsqu'il y a une tentative de sauvegarder l'état de la méthode appelante mais qu'il ne reste plus d'espace dans la pile.

Remarque:

Pour suivre la suite des appels récursifs en NetBeans, vous pouvez procéder comme suit: Placez un point d'arrêt (*breakpoint* ) à côté de l'appel récursif ou à l'intérieur de la méthode récursive.

Démarrez le programme à l'aide de '*Debug Project*' ().

Après que le programme se soit arrêté au point d'arrêt, poussez sur la touche F7 (ou le bouton ) pour traverser le code pas à pas.

A gauche, dans la fenêtre *'Debugging',* vous allez voir la suite des appels récursifs comme ils sont mémorisés sur la pile (EN : **stack**). En bas, dans la fenêtre *'Variables'* vous allez pouvoir suivre le contenu des variables et des paramètres. Vous pouvez même appliquer un double clic sur les différents appels pour voir les valeurs des paramètres lors des appels récursifs.

G. Les structures de données standard

G.1. Tableau - Array

G.1.1. Tableau à une dimension

Les **tableaux**, généralement appelés « **array** », sont des ensembles de données du même type. Les éléments peuvent être des **objets ou des valeurs des types primitifs** (int, double, …). En Java, les tableaux sont déclarés de la manière suivante :

```
<type>[] <name> = { <value_1>, <value_2>, ... , <value_n> };
<type>[] <name> = new <type>[<length>];
<type>[] <name>;
```

- La première ligne déclare un tableau et l'initialise directement avec un certain nombre d'éléments. La taille du tableau est donnée par le nombre d'éléments y inscrits.
- La deuxième ligne déclare un tableau et l'initialise avec autant d'éléments que spécifiés par « **<length>** ». Les éléments contiennent la valeur **null** pour les objets, **0** ou **false** pour les types primitifs.
- La troisième ligne déclare un tableau sans l'initialiser.
- En Java, la taille d'un tableau est fixe. Une fois initialisée, la taille du tableau ne peut plus être changée. Parfois les tableaux sont encore appelés **listes statiques.**
- Le premier élément d'un tableau se trouve toujours à la position « 0 ».
- Tous les tableaux en Java possèdent un attribut « **length** » qui contient le nombre d'éléments du tableau.
- L'accès aux éléments d'un tableau est donné en ajoutant **[***<position>***]** derrière le nom du tableau, où *<position>* est la position à laquelle on veut accéder.
- Comme la taille d'un tableau est fixe, on essaie en général d'estimer le nombre maximal d'éléments utiles à l'application et on réserve l'espace nécessaire lors de l'initialisation. Peu à peu on remplit le tableau du début vers la fin en gardant toujours un bloc continu d'éléments au début du tableau. Ainsi, les **n** premières positions contiennent des éléments, le reste du tableau est vide. Le nombre maximal d'éléments utilisables (c.-à-d. la taille utilisée lors de l'initialisation) s'appelle alors la **dimension maximale**. Le nombre **n** d'éléments qui se trouve actuellement dans un tableau s'appelle la **dimension effective**.

Exemples

```
int[] fibonacci = {1,1,2,3,5,8,13,21,34}
for(int i=0; i<fibonacci.length; i++)
    System.out.println( fibonacci[i] );
```

Supposons l'existence d'une classe « **Voiture** ».

```
Voiture[] voitures = new Voiture[100]; // dimension maximale  = 100
int n=0;                               // dimension effective = 0
voitures[n] = new Voiture();           // création du premier élément
n++;                                   // dimension effective = 1
```

G.1.2. Tableau à deux dimensions

Lors de la définition d'un tableau, le nombre de crochets indique le nombre de dimensions du tableau. Pour définir un tableau à deux dimensions, on emploie la syntaxe suivante :

```
<type>[][] <name> = { { <value_1_1>, <value_1_2>, ... , <value_1_n> },
                      { <value_2_1>, <value_2_2>, ... , <value_2_n> },
                                        . . .
                      { <value_m_1>, <value_m_2>, ... , <value_m_n> } };
<type>[][] <name> = new <type>[<height>][<width>];
<type>[][] <name>;
```

- Les 3 premières lignes déclarent un tableau de m lignes à n colonnes et l'initialisent directement avec un certain nombre d'éléments. La taille du tableau est donnée par le nombre d'éléments y inscrits.
- La 4e ligne déclare un tableau et l'initialise avec autant d'éléments que spécifiés par **<height>** et **<width>**. Les éléments contiennent la valeur **null** pour les objets, **0** ou **false** pour les types primitifs.
- La 5e ligne déclare un tableau sans l'initialiser.

Exemple

```
int[][] tab = new int[5][10] ;
```

définit un tableau à 2 dimensions : 5 lignes sur 10 colonnes (ou l'inverse).

tab[2] désigne le 3e tableau (indice 2) à 10 entiers.

tab[3].length désigne la taille du 4e tableau (indice 3).

Un tableau à plusieurs dimensions peut être initialisé :

```
int[][] tab= {{1,2,3}, {4,5,6}};
```

définit un tableau dont la première dimension va de l'indice 0 à l'indice 1 et la deuxième dimension de l'indice 0 à l'indice 2.

Les différentes lignes d'un tableau à 2 dimensions n'ont pas forcément le même nombre d'éléments :

```
int[][] tab = new int[5][];
for( int i = 0; i<tab.length; i++){
   tab[i]= new int[i+1];
   for( int j = 0; j<tab[i].length; j++){
      tab[i][j] = i*10+j;
   }
}
```

Quel est le contenu de ce tableau ?

G.1.3. Tableaux en paramètre

La spécification d'un paramètre tableau se fait en écrivant autant de couples de [] que le tableau a de dimensions, mais sans indiquer la taille de chaque dimension. La taille peut être obtenue à l'aide de l'attribut **length**.

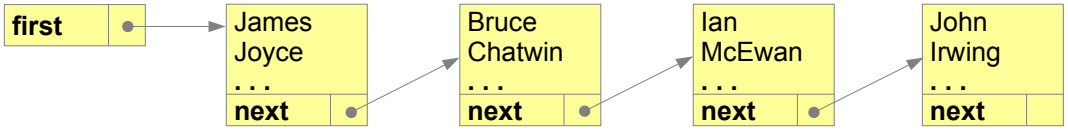
Le contenu des éléments d'un tableau passé comme paramètre peut être modifié, mais pas le tableau lui-même.

G.2. Liste chaînée - Simple Linked List

Principe :

Dans une liste chaînée, chaque objet contient une référence (un pointeur/une adresse) de l'objet suivant. Le dernier objet contient comme référence la valeur **null**. Le début **first** de la liste est un pointeur qui joue un rôle fondamental et qui ne doit pas être 'perdu'. Si la liste est vide, le pointeur **first** est **null**.

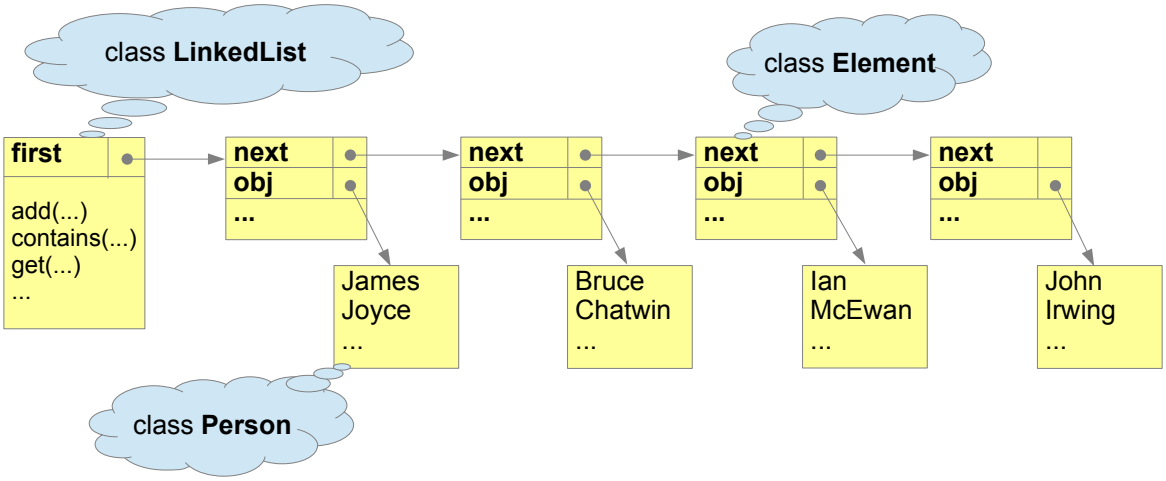
On pourrait imaginer une liste de personnes, où chaque personne pointe sur la prochaine personne :



Dans ce cas il faudrait reprogrammer la logique de la liste chaînée pour chaque type d'objets (personnes, voitures, clients, etc). Ainsi il vaut mieux programmer une structure utilisable plus généralement.

Une telle liste chaînée saura gérer toutes sortes d'objets car elle ne contient pas de données spécifiques, mais une référence aux objets à mémoriser (tout comme **`ArrayList`** et les autres collections prédéfinies.)

Exemple :



Remarques :

- Il est utile, voire indispensable, de représenter graphiquement les objets et les références pendant le développement d'un programme traitant des listes chaînées.
- Pendant les opérations, il est souvent nécessaire d'employer un ou plusieurs pointeurs (variables objet) comme références supplémentaires!
- Si une méthode doit changer une référence, alors nous envoyons la référence comme paramètre et nous retournons la référence modifiée comme résultat. La méthode appelante doit alors affecter la nouvelle valeur à la référence originale.

  P.ex. :

```
myList  = deleteElementByName( myList, "James") ;
```

G.3. Arbre binaire - Binary Tree

La structure d'arbre est une structure de données primordiale en informatique. Nous allons nous concentrer sur la variante des arbres binaires et plus spécialement sur les arbres binaires de recherche.

G.3.1. Vocabulaire

Un **arbre** *(EN : tree)* est une collection d'éléments de même type. Chaque élément de l'arbre est appelé **nœud** *(EN : node)* et le nœud initial est appelé **racine** *(EN : root)*. Chaque nœud stocke une information appelée **valeur** du nœud. Cette valeur peut aussi être une référence à un autre objet porteur d'informations.

Un arbre peut être défini récursivement par :

- un premier nœud (la racine de l'arbre),
- 0, 1, 2 ou plusieurs **sous-arbres** *(EN : sub-trees)*.

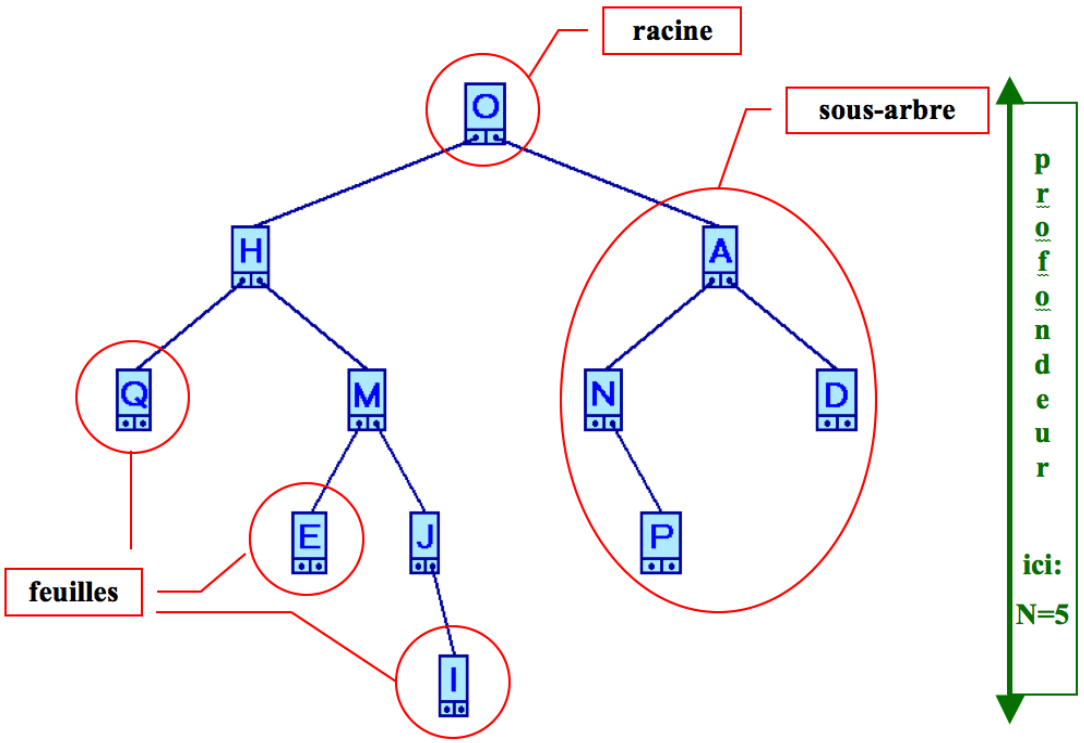
Un nœud sans descendant est appelé une **feuille** *(EN : leaf)* de l'arbre.

On nomme **profondeur** *(EN : depth)* d'un arbre le nombre de niveaux dans un arbre.

Un **arbre binaire** *(EN : binary tree)* est un arbre dans lequel chaque nœud possède 0, 1 ou 2 descendants:

- Chaque nœud a un descendant ***gauche*** et un descendant ***droit*** (éventuellement vides)
- Un arbre binaire peut être vide.

Le nombre maximal de nœuds dans un arbre binaire de profondeur N est donc $\mathbf{2^N-1}$.



G.3.2. Parcours d'un arbre binaire

Souvent, on souhaite visiter chacun des nœuds dans un arbre et y examiner la valeur. Il existe plusieurs ordres dans lequel les nœuds peuvent être visités, et chacun a des propriétés utiles qui sont exploitées par les algorithmes basés sur les arbres binaires.

- Parcours en **préordre** (préfixé) :
  - Visiter la racine (et traiter la valeur)
  - Visiter le sous-arbre de gauche
  - Visiter le sous-arbre de droite
- Parcours en **ordre** (infixé) :
  - Visiter le sous-arbre de gauche
  - Visiter la racine (et traiter la valeur)
  - Visiter le sous-arbre de droite
- Parcours en **postordre** (postfixé) :
  - Visiter le sous-arbre de gauche
  - Visiter le sous-arbre de droite
  - Visiter la racine (et traiter la valeur)

Exemple: Veuillez noter la suite des nœuds si l'arbre de la page précédente est visité

- en préordre: **O, H, Q, M, E, J, I, A, N, P, D**
- en ordre: **Q, H, E, M, J, I, O, N, P, A, D**
- en postordre: **Q, E, I, J, M, H, P, N, D, A, O**

G.3.3. Arbres binaires de recherche - Binary Search Tree

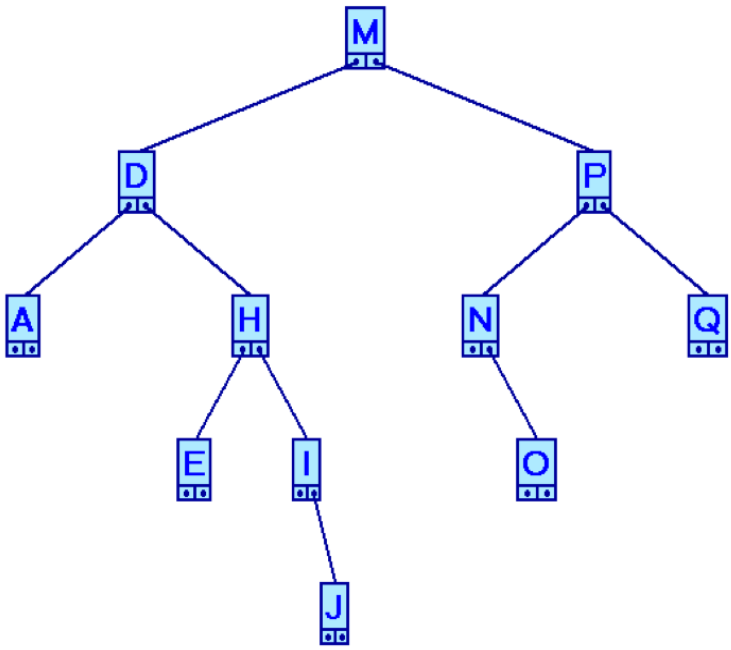
On dit qu'un arbre binaire est un **arbre binaire de recherche** (**ABR**), en anglais **Binary Search Tree** (**BST**), si pour tout nœud de l'arbre, il est vérifié que

- les valeurs du sous-arbre **gauche** du nœud sont **inférieures** à celles du nœud et
- les valeurs du sous-arbre **droit** du nœud sont **supérieures ou égales** à celles du nœud.

Exemple: Si les valeurs de l'exemple ci-dessus sont entrées dans l'ordre

M, D, A, P, N, Q, H, I, J, E, O

dans un ABR, on obtient le résultat suivant :



Si l'arbre de la page précédente est visité *en ordre*, la suite des valeurs est la suivante :

A, D, E, H, I, J, M, N, O, P, Q

On remarque que :

L'arbre est parcouru de gauche à droite et en commençant par la plus petite valeur (à l'extrémité gauche) et en s'arrêtant à la plus grande valeur (à l'extrémité droite). Enfin les valeurs sont traitées par ordre croissant.

G.3.4. Remarques sur les ABR:

Dans le cas général, la **recherche** d'un élément dans un arbre binaire de recherche reste linéaire dans le pire des cas et logarithmique en moyenne. On peut rendre cette recherche logarithmique dans le pire des cas en utilisant les arbres équilibrés et leurs variantes.

La même remarque s'applique à **l'insertion** et à la **suppression** d'éléments dans un arbre binaire.

Une fois l'ABR construit, il n'y a plus qu'à exploiter l'une de ses propriétés pour **trier** l'ensemble des éléments. Il suffit d'effectuer le parcours infixe de l'arbre.

Conclusion: Avantages des ABR

- tri efficace car les données sont maintenues ordonnées
- recherche efficace, inspirée par la dichotomie (le plus efficace pour des arbres complets)

H. Collections & Maps

→ **voir exercices ...**

I. Annexe : Observer et PropertyChangeListener

I.1. Le schéma Observer

Parfois, une action ou le changement d'une propriété n'est pas initiée par le contrôleur ou de la vue, mais d'une classe modèle. Dans ce cas, la classe modèle doit informer la vue et/ou le contrôleur du changement (p.ex. pour actualiser la vue, redessiner les objets, etc.).

En général, les classes modèles ne possèdent pas de lien (référence) vers la vue ou le contrôleur. Pour cette raison, un autre mécanisme est introduit: le schéma *Observer*.

Le principe est que des classes peuvent *s'abonner* à être informées s'il y a un changement dans la classe modèle. Ils sont alors les *"observateurs"* ou *"écouteurs"* de la classe modèle.

I.2. PropertyChangeSupport et PropertyChangeListener

Il y a plusieurs possibilités de réaliser ce mécanisme. Ici, nous allons le réaliser à l'aide d'une instance de la classe ***PropertyChangeSupport*** du paquet ***java.beans***. Cet objet sait gérer une liste d'écouteurs qui sont à informer lors d'un changement.

Dans la classe modèle à observer, nous allons déléguer *(Insert Code.../ Delegate Method...)* les méthodes suivantes de *PropertyChangeSupport* :

addPropertyChangeListener(...)	ajouter un écouteur à la liste des abonnés
removePropertyChangeListener(...)	supprimer un écouteur de la liste des abonnés

La méthode suivante est appelée lors d'un changement :

firePropertyChange(...)	informer tous les écouteurs dans la liste des abonnés qu'un changement a eu lieu

Une classe écouteur doit effectuer les opérations suivantes:

- implémenter l'interface **PropertyChangeListener**, c.-à-d.
- réaliser une méthode :
  public void propertyChange(PropertyChangeEvent pce) { ... }
- et s'abonner comme écouteur en appelant la méthode
  addPropertyChangeListener(*this***)** de la classe modèle observée.

Remarque:

La méthode **firePropertyChange** a trois paramètres qui permettent d'indiquer le nom de la donnée qui a changée, la valeur ancienne ainsi que la nouvelle valeur. La nouvelle valeur doit être différente de l'ancienne valeur (ou *null*).

Le plus souvent, toutes les données de la classe modèle sont à actualiser. Dans ce cas, nous n'avons pas besoin de détailler les changements dans les paramètres. Nous allons intégrer l'appel à **firePropertyChange(...)** dans une méthode que nous appelons **updateListeners()**. La classe modèle n'a qu'à appeler la méthode **updateListeners()** à chaque changement dont les écouteurs doivent être informés.

I.3. Exemple

La classe modèle **Source.java** possède des méthodes **modifyData()** et **generateData()** qui font des changements de données de la classe à des intervalles irréguliers. La classe **MainFrame.java** doit en être informée pour pouvoir s'actualiser.

Source.java :

```
public class Source {

    ...

    //********************** PropertyChangeSupport *************************
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        pcs.addPropertyChangeListener(listener); //ajouter un abonné
    }

    //pas obligatoire, mais parfois utile
    public void removePropertyChangeListener(PropertyChangeListener listener) {
        pcs.removePropertyChangeListener(listener); //supprimer un abonné
    }

    private void updateListeners() { // méthode à appeler à chaque changement
        //Attention: oldValue doit être différent de newValue (ou null)
        pcs.firePropertyChange("SourceData", oldValue, newValue);
    }
   //*************************************************************************
    ...

    public void modifyData(...) {
         ...
         updateListeners(); //informer les abonnés du changement
    }

    public void generateData(...) {
         ...
         updateListeners(); //informer les abonnés du changement
    }
}
```

MainFrame.java :

```
public class MainFrame extends javax.swing.JFrame
                                          implements PropertyChangeListener {

    private Source source = new Source();

    public MainFrame() {
        initComponents();
        source.addPropertyChangeListener(this); //s'abonner
    }

    @Override
    public void propertyChange(PropertyChangeEvent arg0) {
          updateView();  //s'actualiser
    }

    ...
}
```