Pratique du langage Java

1 Réaliser des tests

Les tests de défauts structurels (boite blanche) et fonctionnels (boite noire) - voir le cours cTest.pdf.

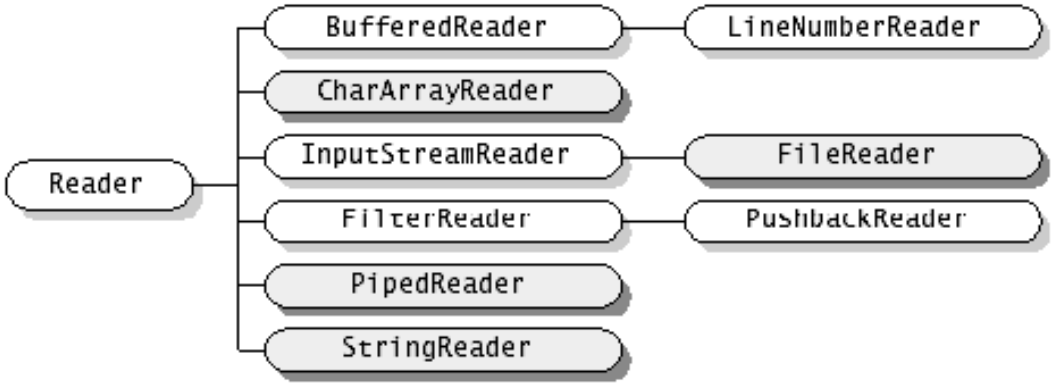
2 Les entrées-sorties

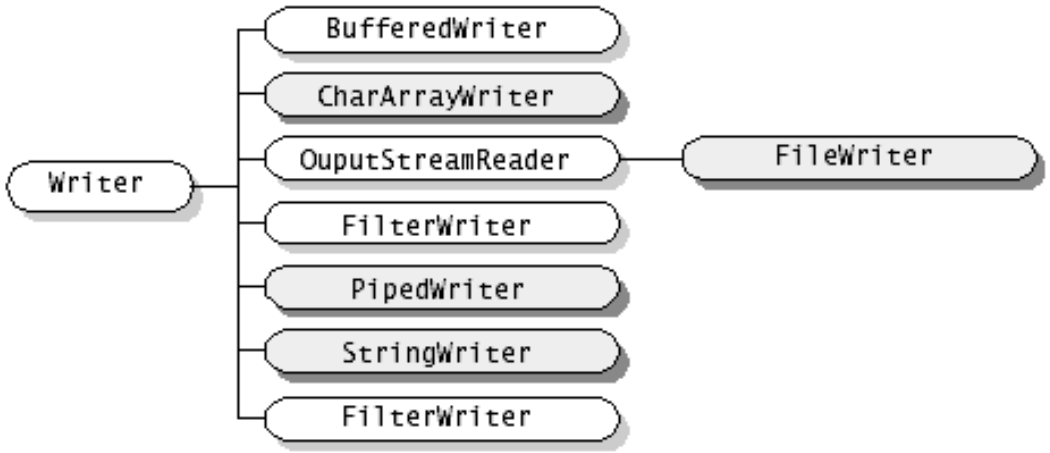
Pour utiliser les classes d'entrées-sorties, il faut importer le *package* java.io.

Flôt (Stream) : collection que l'on peut remplir ou vider en gardant un index sur la dernière position manipulée.

2.1 Flôt de caractères

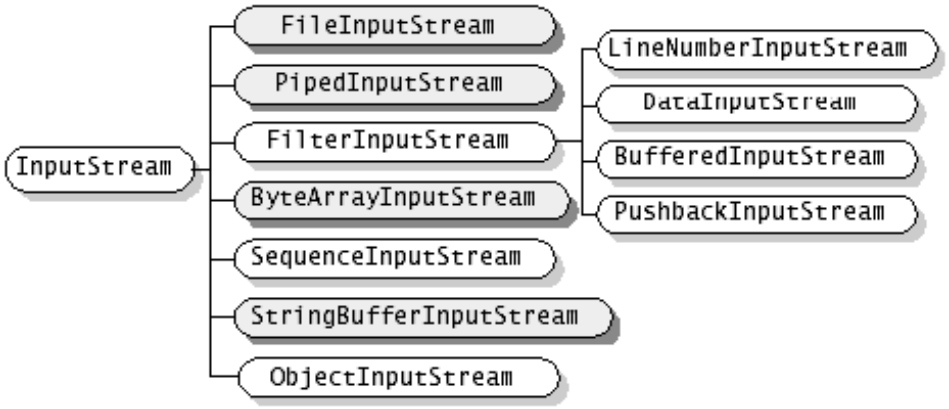
Utilisé typiquement pour lire et écrire des données de type texte.

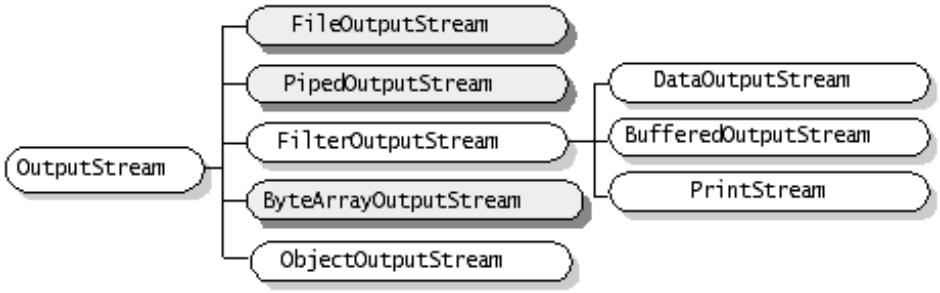

Reader
BufferedReader
LineNumberReader
CharArrayReader
InputStreamReader
FileReader
FilterReader
PushbackReader
PipedReader
StringReader


BufferedWriter
CharArrayWriter
OuputStreamReader
FileWriter
Writer
FilterWriter
PipedWriter
StringWriter
FilterWriter

2.2 Flôt d'octets

Utilisé typiquement pour lire et écrire des données binaires (exécutables, images, sons, ...).


InputStream
FileInputStream
PipedInputStream
FilterInputStream
ByteArrayInputStream
SequenceInputStream
StringBufferInputStream
ObjectInputStream
LineNumberInputStream
DataInputStream
BufferedInputStream
PushbackInputStream


OutputStream
FileOutputStream
PipedOutputStream
FilterOutputStream
ByteArrayOutputStream
ObjectOutputStream
DataOutputStream
BufferedOutputStream
PrintStream

2.3 Entrées sorties standard

Les entrée-sortie standards sont des flots d'octets. Il sont accessibles comme des membres statiques (in, out et err) de la classe java.lang.System.

```
InputStream stdin = System.in;
OutputStream stdout = System.out;
```

2.4 Utilisation de base

Méthodes pour lire des caractères ou des tableaux de caractère sur la classe `Reader`

```
int read()
int read(char cbuf[])
```

Méthodes pour lire des caractères ou des tableaux d'octets sur la classe `InputStream`

```
int read()
int read(byte cbuf[])
```

Exemple, lecture d'un octet sur l'entrée standard et écriture sur la sortie standard

```
try{
   byte b;
   int val = System.in.read();
   if(val != -1)
      b = (byte)val;
   System.out.write(b);
```

```
}
```

On trouve les méthodes d'écriture équivalentes sur les classes `Writer` et `OutputStream`

On trouve de plus des méthodes classiques pour manipuler les flôts (lire sans avancer, marquer une position, remettre à zéro ...).

Tous les flôts sont automatiquement ouverts à leur création. La fermeture peut être soit explicite, soit réalisée par le GC.

2.5 Exemple de lecture/écriture dans un fichier

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

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

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

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

2.6 Composition de flôts

La composition permet de sérier l'application de fonctionnalités.

Exemple

- brancher un flôt capable de lire une ligne dans un fichier sur un flôt standart de lecture de caractères dans un fichier

  ```
  BufferedReader in = new BufferedReader(new FileReader(‘‘f’’));
  String l = in.readLine();
  ```

  La classe `BufferedReader` est dotée de la méthode `readLine`.

- Utilisation de la classe `StreamTokenizer` qui sait lire des mots sur un flux de

caractères.

```
StreamTokenizer in = new StreamTokenizer(new FileReader(‘‘f’’));
??? i = in.nextToken();
...
```

Voir la documentation pour tous les détails de l'utilisation de cette classe.

2.7 Conversions

`InputStreamReader` lit des octets et les transforme en caractères.

Exemple : lire des chaînes de caractères depuis le terminal :

```
StreamTokenizer in =
    new StreamTokenizer(
        new InputStreamReader(System.in))
```

`OutputStreamWriter` convertit les caractères en octets.

3 Les processus concurrents

3.1 La classe Threads et ses sous-classes

Une instance de la classe *Thread* représente un processus.

Cycle de vie : Un processus comprend un certain nombre de messages. Les premiers sont `start` et `stop`. L'envoi du message `start` provoque l'exécution de la méthode `run`, laquelle peut contenir n'importe quelles instructions Java.

Il existe deux façons de réaliser une application dans laquelle seront utilisés des processus concurrents.

3.2 Héritage : Créer une sous-classe de Thread

Une instance I d'une sous-classe C de la classe `Thread` est un processus et comprend donc les messages `start` et `stop`. Redéfinir sur C la méthode `run` permet de spécifier l'activité de I quand il est lancé.

3.3 Créer une sous-classe anonyme de Thread

```
Thread myThread = new Thread() {
        public void run(){
          try{
            // code exécuté dans le processus.
          catch(InterruptedException e){}}};
```

3.4 Composition : Implanter l'interface Runnable

Comme l'héritage est simple, il peut être génant que votre classe soit une sous-classe de `Thread`. Il est alors possible d'utiliser la composition à condition d'implanter l'interface *runnable* qui déclare la méthode `run`.

- Créez votre classe C avec une variable d'instance `myThread` de type *Thread*.
- Faites en sortes que C implémente *Runnable* et définisse donc une méthode `run`.
- Initialisez `myThread` de la façon suivante :

```
mythread = new Thread(this);
```

Le constructeur de la classe `Thread` accepte ceci à condition que l'argument soit de type *Runnable*

- L'instruction `myThread.start()` provoquera alors l'invocation de la méthode `run` de la classe C.

3.5 La synchronisation des accès aux objets

La synchrisation est basée sur des moniteurs qui garantissent l'accès exclusif à un objet via des sections critiques.

- **Synchronisation passive**

  Définir une méthode avec le mot-clé *synchronized* dans la définition de la méthode (exemple, utile pour la méthode `pop` de la classe `Stack`). Le corps de la méthode est toujours utilisé en section critique.

- **Synchronisation active**.

  Il est possible de faire exécuter toute séquence d'instructions en section critique relativement à un objet.

```
synchronized (expr) { statements }
```

L'expression `expr` doit rendre en valeur l'objet concerné par les expressions `statements`. Aucun autre processus ne pourra accéder à cet objet durant l'exécution des instructions.

3.6 Gestion fine des processus

Un ensemble de méthode définies sur la classe classiques de suspendre, de reprendre, etc, l'exécution processus.

- `wait` définie sur la classe Object indique au processus utilisant l'objet receveur de le libérer et d'attendre jusqu'à ce qu'un notify soit exécuté par un autre processus pour cet objet.
- `notify` définie sur Object, réveille un processus en attente sur le receveur sans libérer le receveur immédiatement (l'objet est libéré à la fin de la section critique).
- `sleep`
- ...

4 Les exceptions - Présentation complète

Le modèle de gestion des exceptions est inspiré de ceux de Smalltalk et de C++, eux-même inspirés ...

4.1 Les classes d'exceptions

Chaque sorte d'exception est défini par une classe. Les sortes d'exception sont organisées en hiérarchies.

```
Throwable
  Error
    ...
  Exception
    RuntimeException
    ...
```

4.2 Création d'une nouvelle sorte d'exception

Création d'une sous-classe de *Exception*.

```
abstract class StackException extends Exception {
  Stack s;
  StackException(Stack s2) {s = s2;}
}

class FullStack extends StackException {
  FullStack(Stack s2) {super(s2);}
  void describe() {...}
}
```

4.3 Signalement - Instruction *throw*

```
class Stack {
  int index = 0;
  Object[] buffer = new Object[10];

  void push(Object o) throws FullStack {
    if (index == buffer.length)
      throw new FullStack(this);
    //suite
  }
}
```

Un signalement d'exception provoque une recherche de traitant

(handler) dans la pile d'exécution. L'instance signalée sera passée en argument à tous les handlers.

4.4 Définition d'un handler - instruction *try-catch*

Il peut y avoir n clauses *catch*, une seule sera exécutée. Elle doivent être ordonnées en tenant compte de la hiérarchie des classes d'exceptions.

Notez la possibilité de rattraper différente exceptions avec une seule clause.

```
class testException {
  public static void main(String[] args)
    {Stack testStack = new Stack();
    try {testStack.push(new Integer(1));}
    catch (ExceptionStack f)
          { ... traitement de l'exception ... } } }
```

Durant l'exécution de la clause `catch`, f est liée à l'instance de l'exception signalée.

4.5 Traitement de l'exception

- 1) Signaler une nouvelle exception

- 2) Remettre le système dans un état cohérent et se terminer. L'exécution

continuera alors à l'instruction suivant l'instruction *try*.

4.6 Restaurations inconditionnelles - instruction *try-finally*

```
try { ... actions ...}
finally { ... restaurations ...}
```

Les instructions `restaurations` seront exécutées après l'éxécution des instructions `actions` quoi qu'il arrive durant leur exécution.

5 Réflexivité

Exemple d'utilisation de ce package.

```
import java.lang.Reflect;

public class TestReflect {

    public static void main (String[] args) throws NoSuchMethodException{
        GraphicCptBean g = new GraphicCptBean();

        Class gClass = g.getClass();

        Class gSuperclass = gClass.getSuperclass();

        Method gMeths[] = gClass.getDeclaredMethods();

        Method getCompteur = gClass.getDeclaredMethod("getCompteur", null);

        try {System.out.println(getCompteur.invoke(g, null));}
```

```
        catch (Exception e) {}
    }
}
```

6 Applets

6.1 Définition

Applet: code exécutable dont les résultats calculés et les effets de bord visuels peuvent être affichés dans un page WEB.

Une applet java est une instance d'une sous-classe de la classe `Applet` qui définit sa structure et ses comportements de base.

```
java.lang.Object
  |
  +--java.awt.Component (les composants graphiques)
        |
        +--java.awt.Container (les composants conteneurs de composants)
              |
              +--java.awt.Panel (les conteneurs les plus simples)
                    |
                    +--java.applet.Applet
```

La hiérarchie précédente montre qu'une applet est une sorte de composant

graphique qui hérite en particulier d'une méthode *paint* lui permettant de s'afficher sur un support adéquat.

6.2 Cycle de vie d'une applet

Les quatre méthodes suivantes gèrent le cycle de vie de l'applet et ont une définition par défaut fournie dans la classe Applet.

- init()

  invoqué au chargement ou rechargement,

- start()

  invoqué après l'initialisation ainsi qu'après un *stop* si le l'applet redeviens visible,

- stop()

  invoqué lorsque le document contenant disparait (changement de page),

- destroy()

  invoqué avant la destruction de l'applet.

6.3 L'affichage d'une Applet

La méthode `paint` héritée de `component`, peux être redéfinie pour spécifier le comportement visuel de l'applet.

Cette méthode accepte en argument un objet de type `Graphics`. La classe **Graphics** définie des objets représentant une matrice graphique d'un composant d'interface.

Il est possible de dessiner (pixels) sur cette matrice via des messages approrpriés (`drawLine`, `DrawString`, `FillRect`, etc.

6.4 Applet Simple

Programme.

```
package applets;
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {
  public void paint(Graphics g) {
    g.fillRect(5,5,40,40);
    g.drawString("Hello world!", 50, 25);
  }}
```

Page HTML

```
<HTML>
<HEAD>
<TITLE> A Simple Program </TITLE>
</HEAD>
<BODY>
<h1>Une applet affichant un rectangle:</h1>
<APPLET CODE="applets.HelloWorld.class"
        WIDTH=150 HEIGHT=50
        ALIGH= right VSPACE=30 HSPACE=30>
</APPLET>
</BODY>
</HTML>
```

6.5 Applet avec paramètres

Programme

```
package applets;
import java.applet.Applet;
import java.awt.Graphics;

/* D'apres "Java, de l'esprit à la méthode", Bonjour, Falquet, Guyot,
LeGrand 96 */

public class HelloWorld2 extends Applet {
  String parametre;
  int taille;
  public void init() {
    parametre = this.getParameter("Rect");
    if (parametre == null) taille = 50;
    else taille = Integer.parseInt(parametre); }

  public void paint(Graphics g) {
    g.fillRect(5,5,taille,taille);
    g.drawString("Hello world!", 50, 25);  }}
```

Page HTML

```
<HTML>
<BODY>
<h1>Une applet affichant plusieurs rectangles:</h1>
<APPLET CODE="applets.HelloWorld2.class" WIDTH=150 HEIGHT=50
ALIGN=middle ALIGH= left VSPACE=30 HSPACE=30> <PARAM NAME=Rect VALUE="10">
</APPLET>
<APPLET CODE="applets.HelloWorld2.class" WIDTH=200 HEIGHT=75
ALIGN=middle ALIGH= left VSPACE=30 HSPACE=30> <PARAM NAME=Rect VALUE="20">
</APPLET>
<APPLET CODE="applets.HelloWorld2.class" WIDTH=300 HEIGHT=100
ALIGN=middle ALIGH= left VSPACE=30 HSPACE=30> <PARAM NAME=Rect VALUE="50">
</APPLET>
```

6.6 Restrictions de sécurité

* Applets cannot load libraries or define native methods. Applets can use only their own Java code and the Java API the applet viewer provides. At a minimum, each applet viewer must provide access to the API defined in the java.* packages.

* An applet cannot ordinarily read or write files on the host that is executing it.

* An applet cannot make network connections except to the host that it came from.

* An applet cannot start any program on the host that is executing it.

* An applet can only read certain system properties : java.class.version, java.vendor, java.vendor.url, java.version, os.arch, ...

* ... Voir tutorial

7 Les classes String et StringBuffer

String est la classe des chaînes de caractères constantes et *StringBuffer* celle des chaînes modifiables.

7.1 Création et manipulation

```
String s = "ceci est une chaîne";
String s2 = new String("ceci est une chaîne");
s2.charAt(3);
s + s2;
StringBuffer sb1 = new StringBuffer(5);
StringBuffer sb2 = new StringBuffer("une chaîne");
sb2.insert(4, "Belle ");
```