Cours Programmation des entrées sorties en Java

2ème année GII

2021/2022

Plan général

- Les flux d'entrées/sorties (package java.io)
  - Les flux binaires
  - Les flux caractères
  - Nouveautés java 7 (le package java.nio)
- Persistance d'objets et sérialisation
- Communication réseau : les sockets
- Applications distribuées avec RMI: Remote Method Invocation

Principes et Définitions

- Les entrées / sorties sont organisées en Java autour du concept de flux ou flot (stream)
- Définition :
  - Un flux est un canal de communication entre un lecteur et un rédacteur.
- Les flux sont classifiés en :
  - Flux d'entrée qui comportent des méthodes de lecture ;
  - flux de sortie qui comportent des méthodes d'écriture.
- Un flux d'entrée est connecté a une source, un flux de sortie a une cible.
- La source ou la cible d'un flux peuvent être un fichier, une chaine de caractères, une connexion réseau, un autre flux.

Organisation de l'API

- L'API Java propose une infrastructure de flux basée sur les packages java.io et java.nio.
- Les flux sont des objets
- Tous les flux sont des descendants des classes (abstraites) **InputStream, OutputStream**, **Reader et Writer (java1.1)**.
- Les flux élémentaires sont des flux d'octets (flux binaires).
- Les classes pour les flux non structurés (séquences d'octets) en écriture, respectivement lecture, sont OutputStream respectivement InputStream.
- Les classes pour les flux de type séquences de caractères Unicode, en écriture, respectivement lecture, sont Writer et respectivement Reader.
- Les classes InputStreamReader et OutputStreamWriter sont des ponts entre les flux structurés et non structurés.

Organisation de l'API

- Java 1.0 java.io

– InputStream/OutputStream (flux binaires), IOException

- Java 1.1

– Reader/Writer (flux caractères)

- Java 1.4 java.nio.[buffer, charset]

– ByteBuffer, Charset, Channel

- Java 1.7 java.nio.file

– Path, Files

– try-with-resources

Notion de fichier

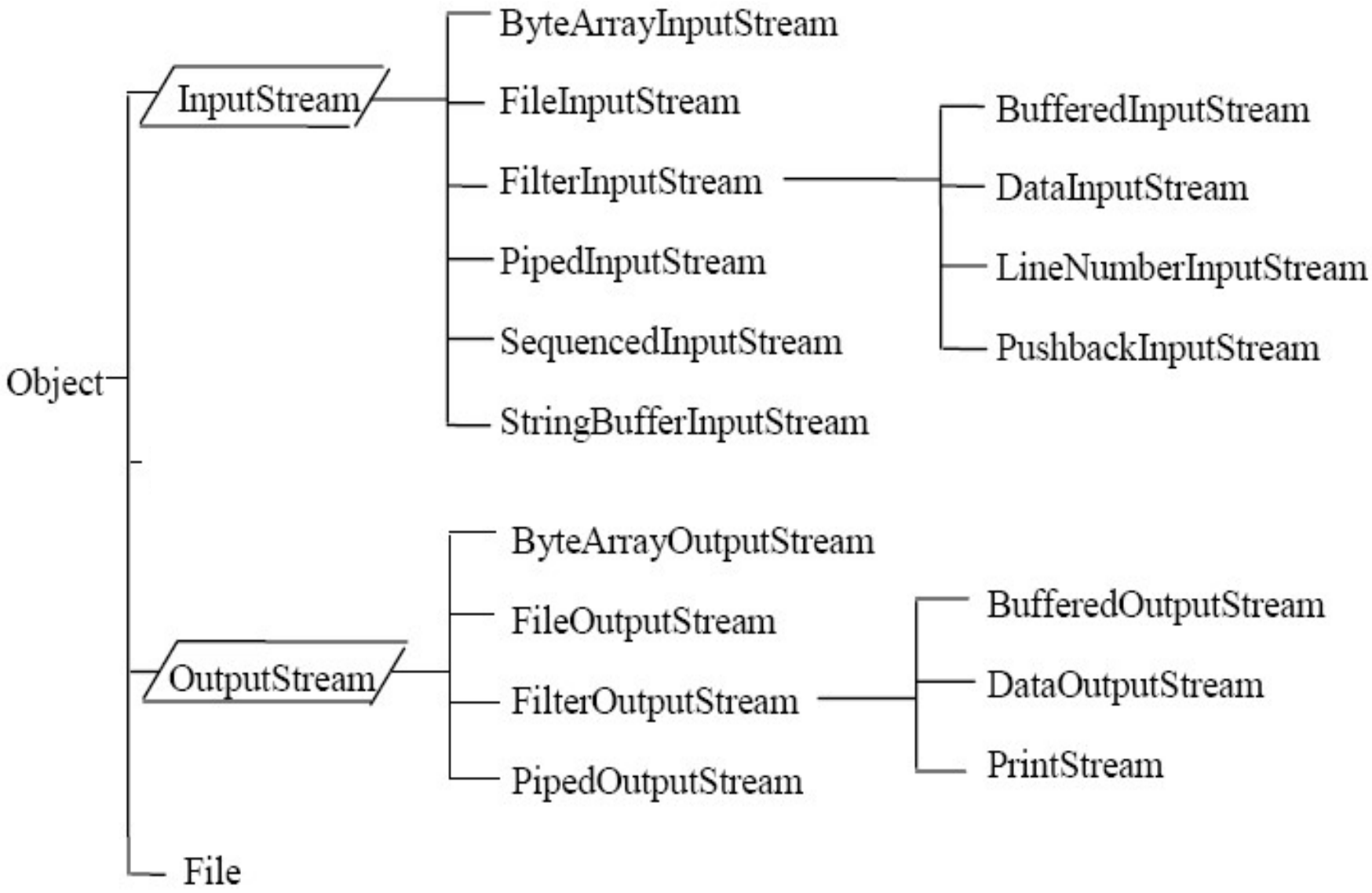
- Définition : un fichier sera ici défini comme un flux enregistré sur un disque.
- Un fichier : objet du système de gestion de fichiers, il a une taille, des droits, une date de création, il appartient à un répertoire... par ailleurs, il contient un flux.
- En java, un fichier est représenté par un objet **File (java 1.6)** ou **Path (à partir de 1.7).**

Quelques méthodes de la classe File

- **File** (String name)
- **File** (String path, String name)
- **File** (File dir, String name)
- boolean **isFile**( ) / boolean **isDirectory**( )
- boolean **mkdir**( )
- boolean **exists**( )
- boolean **delete**( )
- boolean **canWrite**( ) / boolean **canRead**( )
- File **getParentFile**( )
- long **lastModified**( )

Méthode	Type retour	Description
`canRead()`	`Boolean`	le fichier ou le répertoire est-il accessible en lecture ?
`canWrite()`	`Boolean`	le fichier ou le répertoire est-il accessible en écriture ?
`createNewFile()`	`Boolean`	Crée un nouveau fichier
`delete()`	`Boolean`	détruit le fichier ou le répertoire
`exists()`	`Boolean`	est-ce que le fichier ou le répertoire existe ?
`getAbsolutePath()`	`String`	renvoie le chemin absolu du fichier ou du répertoire
`getName()`	`String`	renvoie le nom du fichie ou du répertoire
`getParent()`	`String`	renvoie le nom du répertoire parent du fichier ou du répertoire
`getPath()`	`String`	renvoie le chemin du fichier ou du répertoire
`isAbsolute()`	`Boolean`	le chemin du fichier ou du répertoire est-il absolu
`isDirectory()`	`Boolean`	s'agit-il d'un répertoire ?
`isFile()`	`Boolean`	s'agit-il d'un fichier ?
`lastModified()`	`long`	renvoie la date de la dernière modification du fichier ou du répertoire
`length()`	`long`	renvoie la taille du fichier
`list()`	`String []`	renvoie la liste des fichiers du répertoire
`listfiles()`	`File []`	renvoie la liste des fichiers du répertoire sous la forme d'un tableau d'objet de type `File`
`mkdir()`	`Boolean`	crée un répertoire
`mkdirs()`	`Boolean`	crée tous les répertoires du chemin
`renameTo(File dest)`	`Boolean`	renome le fichier ou le répertoire
`setLastModified()`	`Boolean`	définit l'heure de la dernière modification du fichier ou du répertoire
`setReadOnly()`	`Boolean`	définit le fichier en lecture seule
`toURL()`	`java.net.URL`	génère un objet URL pour ce fichier ou répoertoire

Flux binaire



Exemple : La classe File

```
import java.io.File;

public class Main {
  public static void main(String[] args) {
    //Création de l'objet File
    File f = new File("test.txt");
    System.out.println("Chemin absolu du fichier : " + f.getAbsolutePath());
    System.out.println("Nom du fichier : " + f.getName());
    System.out.println("Est-ce qu'il existe ? " + f.exists());
    System.out.println("Est-ce un répertoire ? " + f.isDirectory());
    System.out.println("Est-ce un fichier ? " + f.isFile());}}
```

```
Chemin absolu du fichier : C:\workspace\File\test.txt
Nom du fichier : test.txt
Est-ce qu'il existe ? true
Est-ce un répertoire ? false
Est-ce un fichier ? true
```

Les flux standard : la console et le clavier

- La classe **System est pourvue de trois attributs de classe :**
  - **in : un flux qui correspond à l'entrée standard (par défaut le clavier)**
  - **out : un flux qui correspond à la sortie standard (par défaut la console)**
  - **err : un flux qui correspond à la sortie d'erreur (par défaut la console)**
- *// Lecture sur le clavier*

  int x = System.in.read() ;
- *// Écriture sur la console*

  System**.out.write(x);**
- Les sorties out et err utilisent un filtre PrintStream qui permet l'utilisation des fonctions print et println.

Flux binaires et flux texte

- un flux binaire est une suite d'octets. L'interprétation de ces octets est à la charge du programme qui les lit ou les écrit.
- un flux texte est une suite de caractères. Il est stocké selon un système de codage déterminé.
- un flux texte est toujours construit au dessus d'un flux binaire.

Les fichiers

On fait généralement la distinction entre deux catégories de fichiers :

– les fichiers dits textuels (text files), qui contiennent une séquence de caractères encodés, et

– les fichiers dits binaires (binary files), qui sont tous les autres fichiers : images, sons, formats propriétaires utilisés par différents programmes, etc.

- tout fichier, même s'il contient du texte, est composé d'une suite d'octets et est donc « binaire »…

La classe InputStream

- Les trois variantes de la méthode read permettent de lire un ou plusieurs octets :

– int read() : lit et retourne le prochain octet sous la forme d'une valeur comprise entre 0 et 255 inclus, ou –1 si la fin du flot a été atteinte,

– int read(byte[] b, int o, int l) : lit au plus l octets du flot, les place dans le tableau b à partir de la position o et retourne le nombre d'octets lus,

– int read(byte[] b) ou read(b, 0, b.length)

- La méthode skip permet d'ignorer des octets

long skip(long n) : ignore au plus n octets du flot et retourne le nombre d'octets ignorés

Lecture bloquante

- Les méthodes read et skip sont potentiellement bloquantes (blocking).
- Cela signifie que si la totalité des octets demandé à ces méthodes n'est pas encore disponible mais le sera dans le futur par exp après avoir été lus du disque ou reçus du réseau, le programme est bloqué jusqu'à l'arrivée des octets.
- Cette caractéristique facilite la programmation mais peut nuire aux performances ou à l'interactivité de l'application.
- L'un des buts de java.nio est de fournir des méthodes permettant l'accès non bloquant aux données.

Lecture bloquante

- Pour éviter de bloquer le programme, il peut être utile de connaître le nombre d'octets qu'il est possible d'obtenir du flot sans blocage. C'est le but de la méthode suivante :

– int available() : retourne une estimation du nombre d'octets qu'il est possible de lire ou d'ignorer sans bloquer.

La classe OutputStream

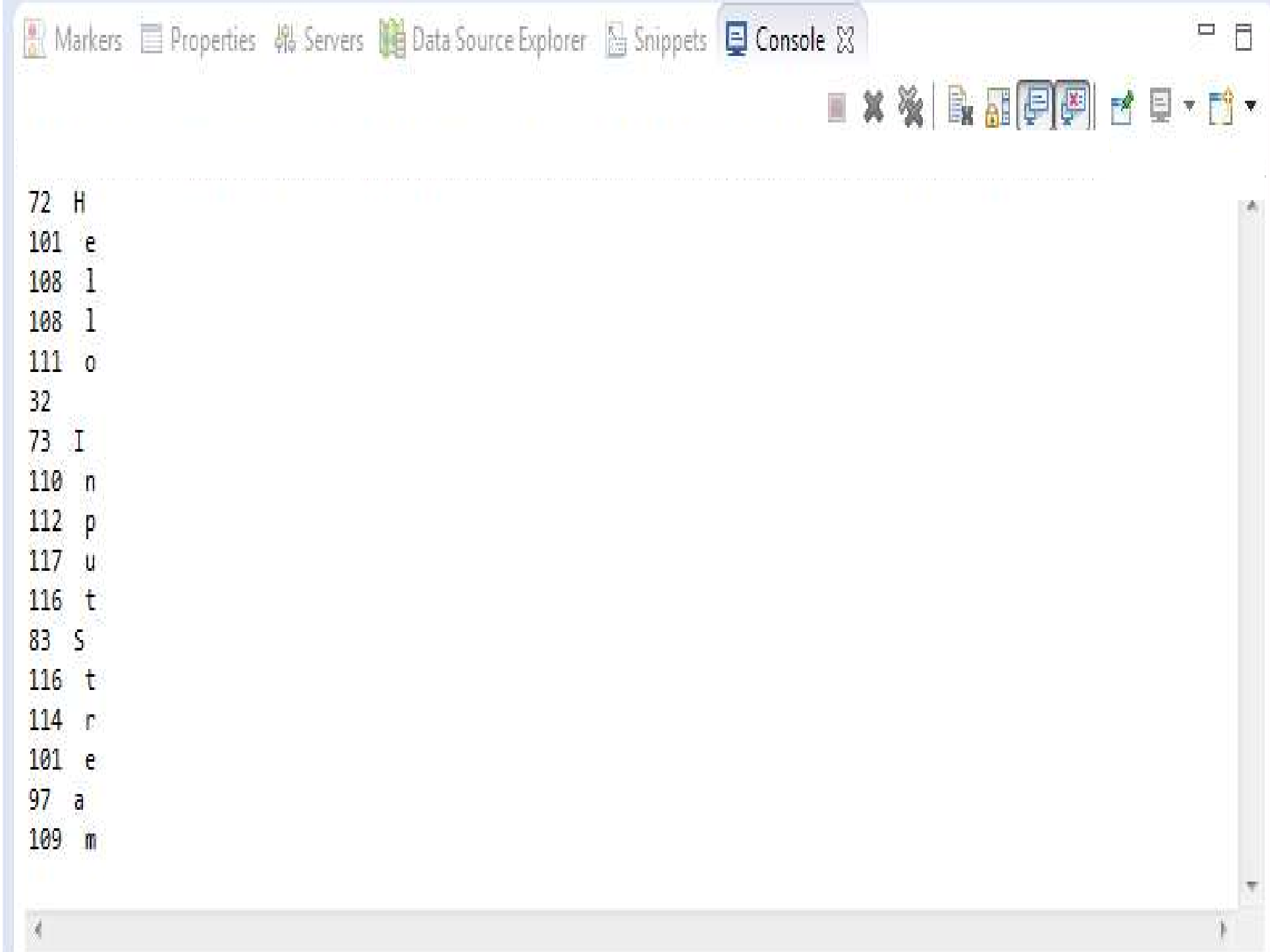
- Les méthodes principales qui peuvent être utilisées sur un OutputStream sont :
  - **public abstract void write (int) throws IOException** qui écrit l'octet passé en paramètre,
  - **void write (byte[] b)** qui écrit les octets lus depuis un tableau d'octets,
  - **void write (byte[] b, int off, int len)** qui écrit les octets lus depuis un tableau d'octets à partir d'une position donnée et sur une longueur donnée,
  - **void close ()** qui permet de fermer le flux après avoir éventuellement vidé le tampon de sortie,
  - **flush ()** qui permet de purger le tampon en cas d'écritures bufferisées

Ouverture/création du flux en écriture/lecture

- **FileOutputStream (File file)**throws IOException
- FileOutputStream (String name) throws IOException
- FileOutputStream (String name, boolean append) throws FileNotFoundException

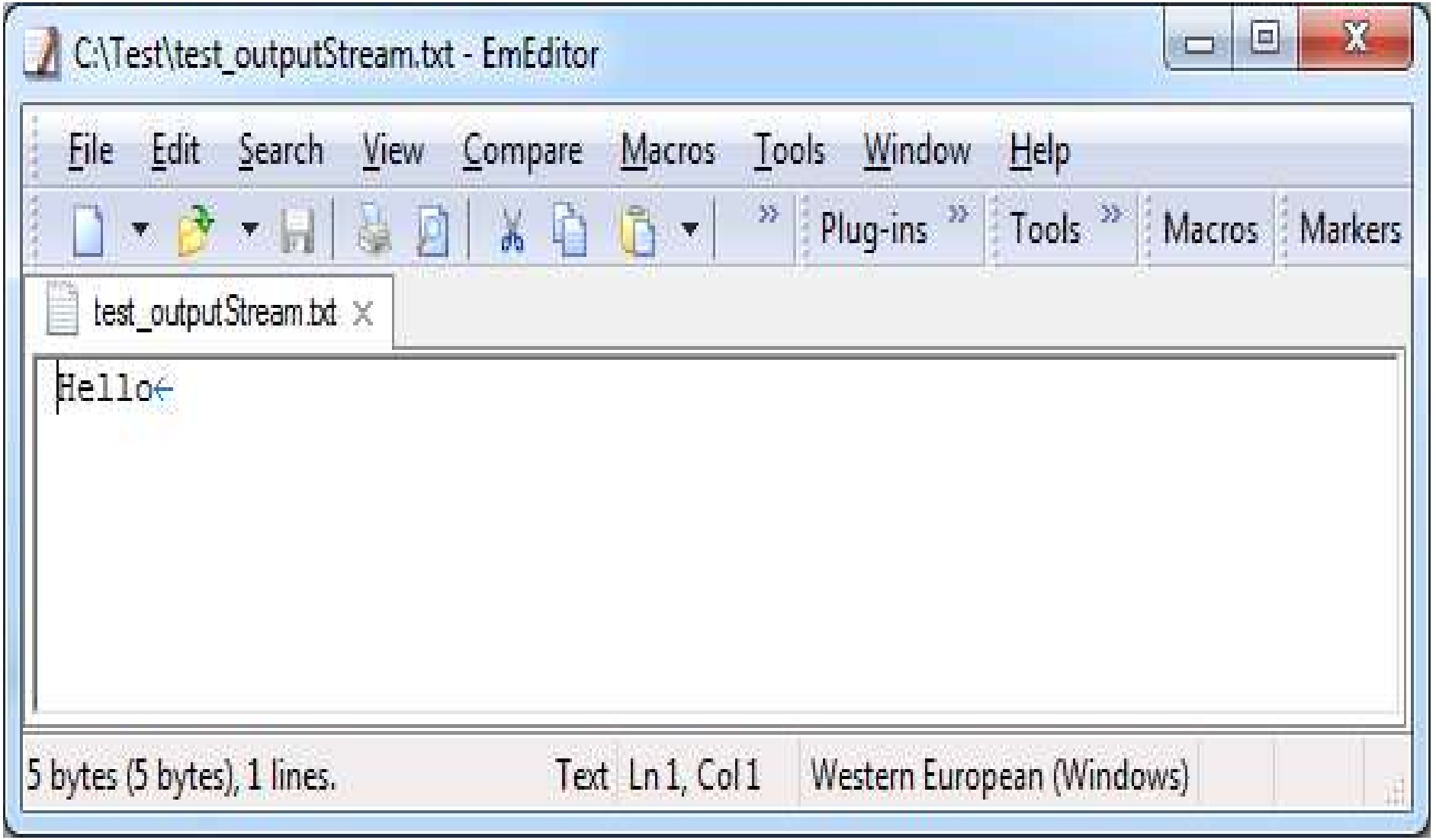
  ouvre le flux en écriture ; ajoute les données écrites à la fin du fichier actuel si append vaut vrai.
- **FileInputStream (File file) throws FileNotFoundException**
- FileInputStream (String name) throws FileNotFoundException

```
import java.io.FileInputStream;
import java.io.InputStream;
public class HelloInputStream {

    public static void main(String[] args) {
        try {
             InputStream is = new FileInputStream("data.txt");
            int i = -1;
           // Read the bytes in the stream.
           // Each time the 8-bit read, convert it to int.
           // Read the value of -1 means the end of the stream.
           while ((i = is.read()) != -1) {
              System.out.println(i + "  " + (char) i);
           }
           is.close();
        } catch (Exception e) {
           e.printStackTrace();
        } }}
```

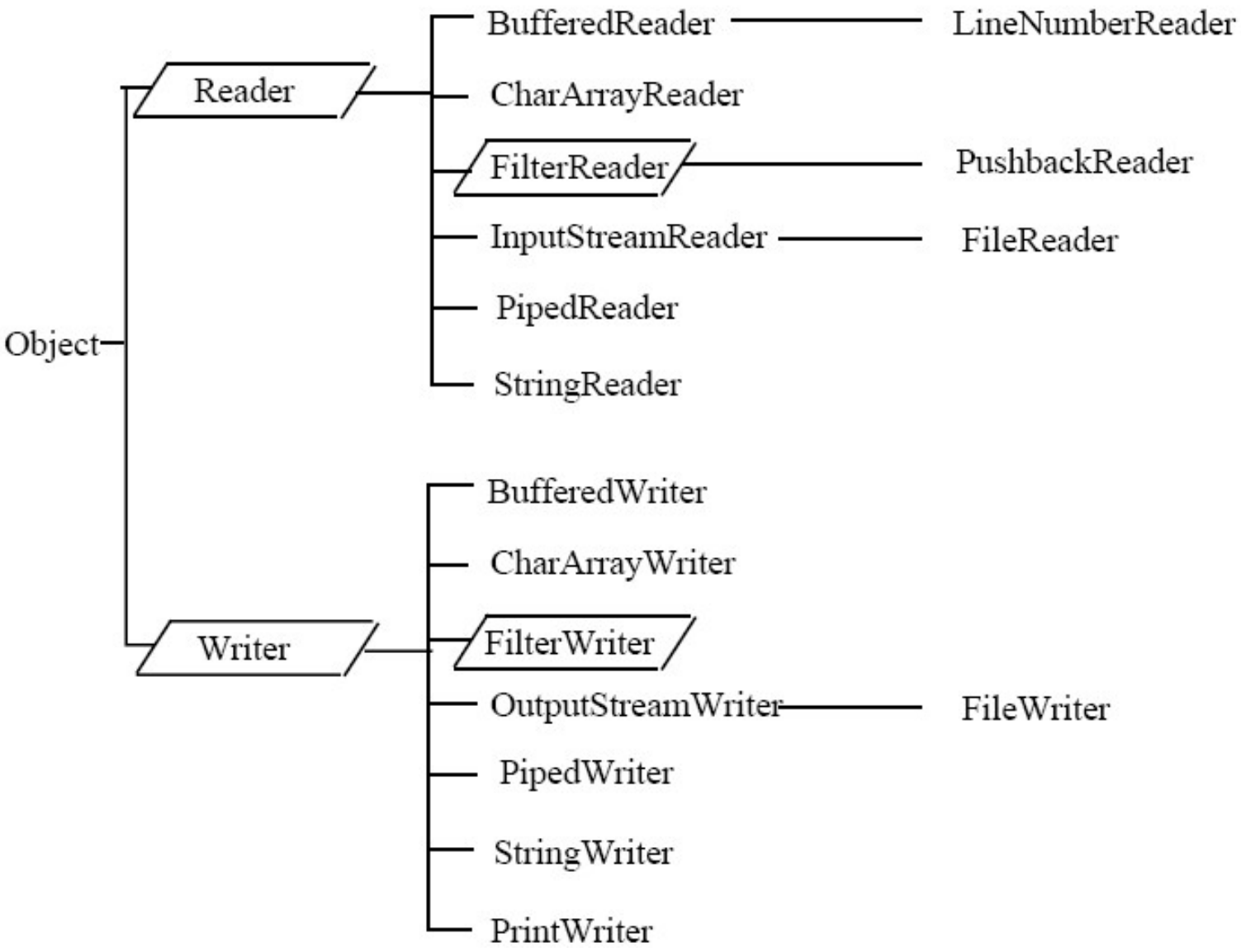

Markers
Properties
Servers
Data Source Explorer
Snippets
Console
72 H
101 e
108 l
108 l
111 o
32
73 I
110 n
112 p
117 u
116 t
83 S
116 t
114 r
101 e
97 a
109 m

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;

public class HelloOutputStream {

   public static void main(String[] args) {
      try {
         // Create output Stream
         //write data to file.
         OutputStream w = new
FileOutputStream(
                  "C:\Test\test_outputStream.t
xt");
         // Create array of bytes, write it to
stream.
         byte[] by = new byte[] { 'H', 'e', 'l',
'l', 'o' };
```

```
 // write the bytes into the
stream
         for (int i = 0; i < by.length;
i++) {
            byte b = by[i];

            // Write 1 byte.
            w.write(b);
         }
         // Close the output
stream, finish write file.
         w.close();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```


C:\Test\test_outputStream.txt - EmEditor
File Edit Search View Compare Macros Tools Window Help
Plug-ins Tools Macros Markers
test_outputStream.txt
Hello
5 bytes (5 bytes), 1 lines.
Text Ln 1, Col 1
Western European (Windows)

La hiérarchie des flux de caractères



La classe Writer

- void **write (int c)** throws **IOException**

  écrit le caractère dont le code est c. Cette méthode fonctionne correctement que c soit un char ou un int.
- void **write (char[] s)** throws **IOException**

  écrit le contenu de s.
- void **write (char[] s, int off, int longueur)** throws **IOException**

  écrit les caractères de s compris entre l'indice off et off + longueur.

La classe Writer

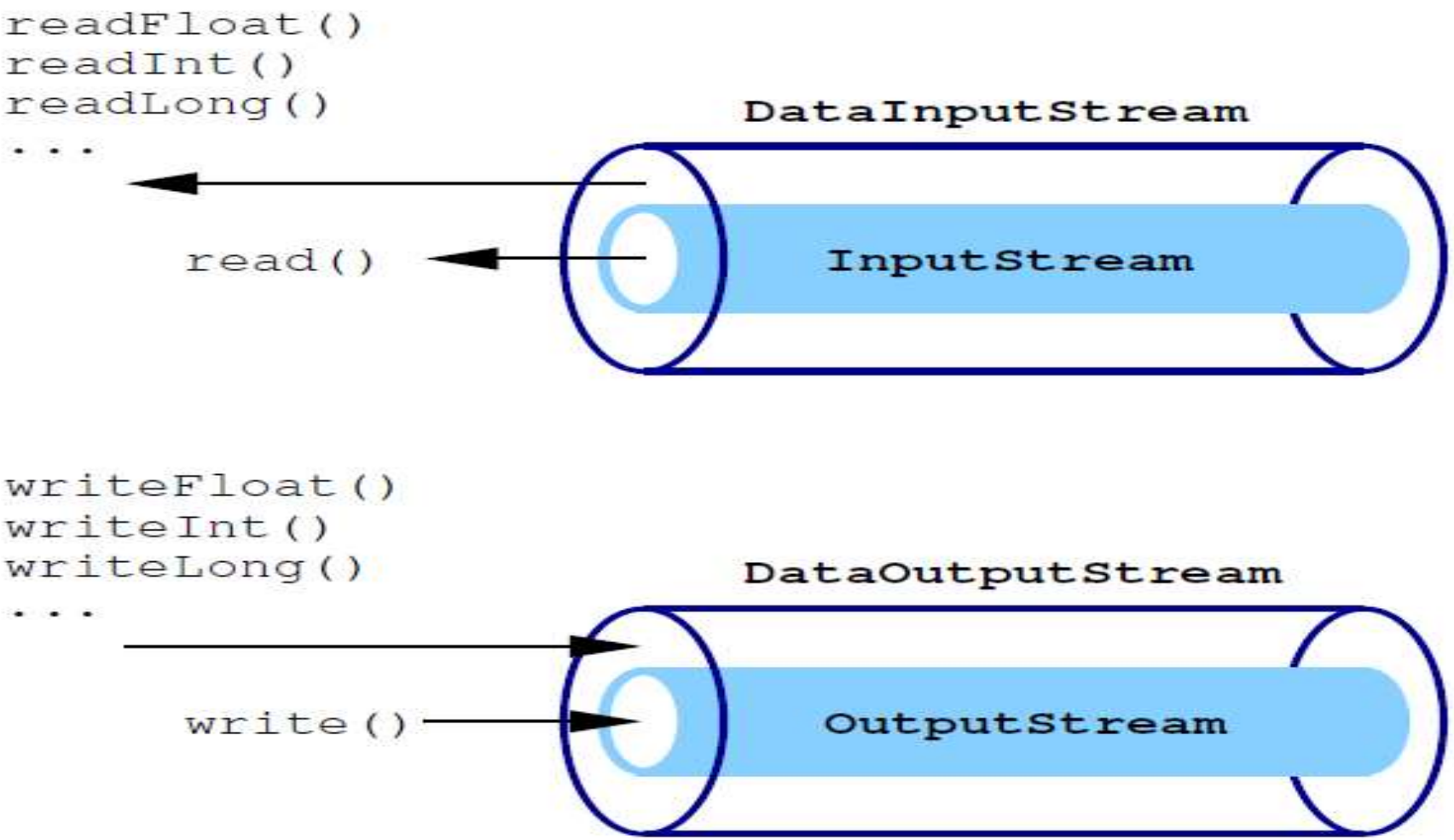
- void **write (String s)** throws **IOException**

  écrit la chaine s.
- void **flush ()** throws **IOException**

  réalise effectivement la copie sur le support. C'est utile dans le cas de classes comme BufferedWriter, ou les données sont stockées temporairement en mémoire.
- void **close ()** throws **IOException**

  ferme le Writer

Les types primitifs



Les enveloppes de flux

- Ce principe consiste à ajouter des fonctionnalités au flux.
- Le plus souvent il s'agit de transformations ou de filtrage.
- Un flux peut prendre le flux cible comme argument de son constructeur et lui délègue les appels après avoir effectué ses opérations de filtrage

Les flux de caractères

- Pour écrire des chaînes de caractères et des nombres sous forme de texte
  - on utilise la classe **PrintWriter** qui possède un certain nombre de méthodes **print (...)** et **println (...)**.
- Pour lire des chaînes de caractères sous forme texte, il faut utiliser, par exemple,
  - **BufferedReader** qui possède une méthode **readLine()** .

Pont entre les flux de caractères et d'octets

- InputStreamReader et OutputStreamWriter sont des flux de caractères enveloppant un flux d'octets.
- Un schéma d'encodage permet une conversion dans les deux sens.
- Le modèle d'encodage peut être donné en paramètre au constructeur.

```
try {
 InputStreamReader convert = new InputStreamReader ( System .in);
// on enveloppe dans BufferedReader pour bénéficier de readLine ()
BufferedReder in = new BufferedReader (convert );
String test =in.readLine ();
}
 catch ( IOException e) {...}
```

Copie en utilisant un buffer

```
public static void copy(InputStream in, OutputStream out)
throws IOException {
byte[] buffer = new byte[8192];
int size;
while((size = in.read(buffer)) != -1) {
out.write(buffer, 0, size);
}
}
```

- La méthode existe déjà

```
inputStream.transferTo(outputStream);
```

Exemples

```
FileInputStream fis = new FileInputStream(new
File("toto.txt"));

BufferedInputStream bis = new BufferedInputStream(fis);
Ou
BufferedInputStream bis = new BufferedInputStream(new
FileInputStream(new File("toto.txt")));

Ou : BufferedInputStream  bis = new BufferredInputStream(
                        new DataInputStream(
                          new FileInputStream(
                            new File("toto.txt"))));
```

Exemple avec buffer

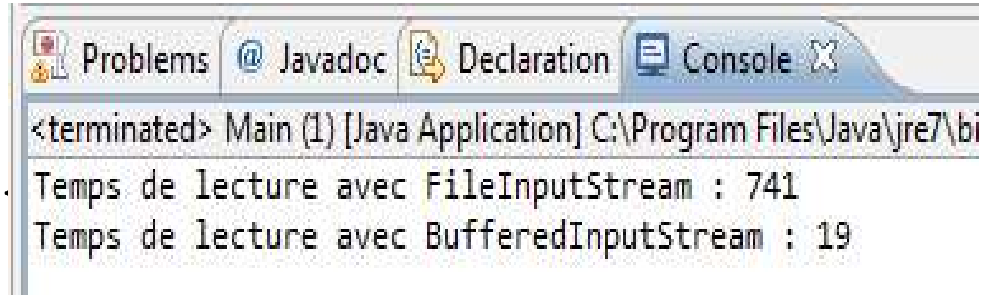
```
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
    //Nous déclarons nos objets en dehors du bloc try/catch
    FileInputStream fis;
    BufferedInputStream bis;
```

Exemple avec buffer

```
try {
    fis = new FileInputStream(new File("test.txt"));//taille fichier 3,6 Mo
    bis = new BufferedInputStream(new FileInputStream(new
File("test.txt")));
    byte[] buf = new byte[8];
    //On récupère le temps du système
    long startTime = System.currentTimeMillis();
    //Inutile d'effectuer des traitements dans notre boucle
    while(fis.read(buf) != -1);
    //On affiche le temps d'exécution
    System.out.println("Temps de lecture avec FileInputStream : " +
(System.currentTimeMillis() - startTime));
```

Exemple avec buffer

```
 //On réinitialise
     startTime = System.currentTimeMillis();
     //Inutile d'effectuer des traitements dans notre boucle
     while(bis.read(buf) != -1);
     //On réaffiche
     System.out.println("Temps de lecture avec BufferedInputStream : " +
System.currentTimeMillis() - startTime));
     //On ferme nos flux de données
     fis.close();
     bis.close();
    } catch (FileNotFoundException e) {
     e.printStackTrace();
    } catch (IOException e) {
     e.printStackTrace();
    }  }}
```



Exemple avec Data(In/Out)putStream

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class Main {
  public static void main(String[] args) {
  DataInputStream dis;
    DataOutputStream dos;
    try {
      dos = new DataOutputStream(
            new BufferedOutputStream(
              new FileOutputStream(
                new File("test.txt"))));
```

```
//Nous allons écrire chaque type primitif
    dos.writeBoolean(true);
    dos.writeByte(100);
    dos.writeChar('C');
    dos.writeDouble(12.05);
    dos.writeFloat(100.52f);
    dos.writeInt(1024);
    dos.writeLong(123456789654321L);
    dos.writeShort(2);
    dos.close();
  dis = new DataInputStream(
            new BufferedInputStream(
              new FileInputStream(
                new File( "test.txt"))));
    System.out.println(dis.readBoolean());
    System.out.println(dis.readByte());
    System.out.println(dis.readChar());
    System.out.println(dis.readDouble());
    System.out.println(dis.readFloat());
    System.out.println(dis.readInt());
    System.out.println(dis.readLong());
    System.out.println(dis.readShort());
   } catch (FileNotFoundException e) {
    e.printStackTrace();
   } catch (IOException e) {
    e.printStackTrace();
   } }}
```

Exemple Flux caractères

(lire et écrire dans un fichier texte)

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class Main {
  public static void main(String[] args) {
    File file = new File("testFileWriter.txt");
    FileWriter fw;
    FileReader fr;
    try {
      //Création de l'objet
      fw = new FileWriter(file);
      String str = "Bonjour à tous !\n";
      str += "\tComment allez-vous ? \n";
       fw.write(str);
         fw.close();
      //Création de l'objet de lecture
      fr = new FileReader(file);
      str = "";
      int i = 0;
      //Lecture des données
      while((i = fr.read()) != -1)
        str += (char)i;
      //Affichage
      System.out.println(str);
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

L'API NIO

- Depuis les débuts de Java, l'API java.io est essentiellement composée de classes et d'interfaces pour réaliser des opérations sur les flux d'octets ou de caractères. Seule la classe File permet des opérations sur les fichiers et les répertoires du système de fichiers.
- L'API NIO a été introduite dans Java 1.4 : elle propose entre autres l'utilisation de channels, buffers et charsets notamment pour permettre de réaliser des opérations de lectures/écritures non bloquantes (non blocking I/O).

Java.nio (JDK 1.4)

- Depuis le JDK 1.4, un nouveau package a vu le jour, visant à améliorer les performances des flux, buffers, etc. traités par java.io (version 1.1 du JDK).
- Nio signifie « New I/O » : ce package a été créé afin d'améliorer les performances sur le traitement des fichiers, du réseau et des buffers. Il permet de lire les données d'une façon différente.
- les objets du package java.io traitaient les **données par octets**. Les objets du package java.nio, eux, les traitent **par blocs de données** : la lecture est donc accélérée.

Java.nio (JDK 1.4)

- Tout repose sur deux objets de ce nouveau package : les channels et les buffers.
- Les channels sont des flux, tout comme dans l'ancien package, mais ils sont amenés à travailler avec un buffer dont vous définissez la taille.
- lorsque vous ouvrez un flux vers un fichier avec un objet FileInputStream, vous pouvez récupérer un canal vers ce fichier. Celui-ci, combiné à un buffer, vous permettra de lire votre fichier encore plus vite qu'avec un BufferedInputStream
- Ce package offre un buffer par type primitif pour la lecture sur le channel : IntBuffer;CharBuffer;ShortBuffer;ByteBuffer;DoubleBuffer;FloatBuffer;LongBuffer.

Le paquetage java.nio.charset

- contient un ensemble de classes permettant de travailler avec des encodages de caractères.
- la classe Charset représente un encodage de caractères, et celui utilisé par défaut peut être obtenu au moyen de la méthode statique defaultCharset
- la classe StandardCharsets de ce paquetage donne accès, via des attributs statiques, aux encodages les plus utiles en pratique : ASCII, ISO 8859–1, UTF–8 et trois variantes de UTF–16

java.nio.charset.StandartCharsets

java.nio.charset.StandartCharsets définie l'ensemble des Charsets par défaut

US-ASCII: seven-bit ASCII,

ISO-8859-1: ISO Latin Alphabet

UTF-8: eight-bit UCS Transformation Format

UTF-16BE Sixteen-bit UCS Transformation Format,

UTF-16LE Sixteen-bit UCS Transformation Format,:

UTF-16: sixteen-bit UCS Transformation Format,

(UTF signifie Unicode Transformation Format)

Représentation des caractères

- Comme n'importe quelle donnée, les caractères doivent être représentés sous forme binaire pour pouvoir être manipulés par un ordinateur.
- En théorie, comme il n'existe qu'un nombre fini de caractères, leur représentation est simple : il suffit d'en établir une liste exhaustive et de représenter chaque caractère par son index dans cette liste, appelé son code.
- En pratique, au vu du grand nombre de caractères existant dans le monde, et de l'invention de nouveaux caractères comme les émoticônes, la simple définition de cette liste est une tâche considérable, actuellement encore en cours.

La norme ASCII

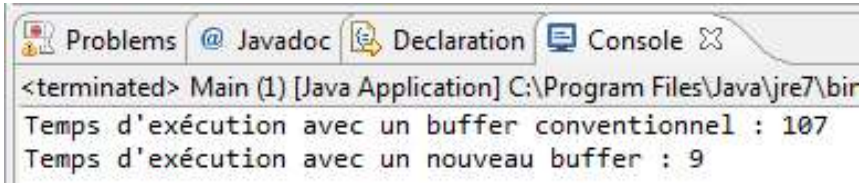
- La norme ASCII (American Standard Code for Information Interchange) représente un caractère par un entier de 7 bits, et permet donc d'en représenter 27 = 128 différents.
- La norme ASCII ne permet pas de représenter tous les caractères de toutes les langues utilisant l'alphabet latin. Par exemple, elle n'inclut pas les lettres accentuées, les ligatures (p.ex. œ), le symbole monétaire de l'Euro (€), etc.
- De nombreuses extensions d'ASCII à 8 bits ont donc été inventées, utilisant la plage des valeurs de 128 à 255 pour ces caractères manquants.

Le standard Unicode

- Le standard Unicode a pour but d'être universel, c-à-d de permettre la représentation de la totalité des « caractères » existants, y compris les symboles graphiques et mathématiques, les émoticônes, etc.

Exemple java.nio lecture fichier

```
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.FileChannel;

public class Main {
  public static void main(String[] args) {
    FileInputStream fis;
    BufferedInputStream bis;
    FileChannel fc;
    try {
      fis = new FileInputStream(new File("test.txt"));
      bis = new BufferedInputStream(fis);
      //Démarrage du chrono
      long time = System.currentTimeMillis();
      //Lecture
      while(bis.read() != -1);
      //Temps d'exécution
      System.out.println("Temps d'exécution avec un buffer
conventionnel : " + (System.currentTimeMillis() - time));
fis = new FileInputStream(new File("test.txt"));
fc = fis.getChannel();
       int size = (int)fc.size();
      //On crée un buffer correspondant à la taille du fichier
ByteBuffer bBuff =ByteBuffer.allocate(size);
      //Démarrage du chrono
      time = System.currentTimeMillis();
      //Démarrage de la lecture
      fc.read(bBuff);
bBuff.flip();
      //Affichage du temps d'exécution
      System.out.println("Temps d'exécution avec un
nouveau buffer : " + (System.currentTimeMillis() - time));

byte[] tabByte = bBuff.array();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace(); }}}
```



L'API NIO 2

- L'API NIO 2 est une API plus moderne qui propose plusieurs caractéristiques :
- la séparation des responsabilités : un chemin (Path) représente un élément du système de fichiers (FileSystem) stocké dans un système de stockage (FileStorage) et est manipulé en utilisant la classe Files
- la gestion de toutes les erreurs se fait avec des exceptions
- NIO 2 repose sur plusieurs classes et interfaces dont les principales sont :
  - Path : encapsule un chemin dans le système de fichiers
  - Files : contient des méthodes statiques pour manipuler les éléments du système de fichiers

L'interface Path

- L'interface Path décrit les fonctionnalités d'une classe qui encapsule un chemin sur le système de fichiers. Ce chemin est représenté sous la forme d'une séquence de noms qui compose la hiérarchie des répertoires du chemin.
- elle encapsule tout ou partie d'un chemin vers un élément du système de fichiers de manière dépendante du système d'exploitation sous-jacent.
- Le chemin peut concerner plusieurs types d'éléments :
  - Un fichier
  - Un répertoire

L'interface Path

- Ce chemin peut être absolu (le chemin contient une racine) ou relatif (en combinaison avec le chemin courant pour obtenir le chemin absolu). La représentation d'un chemin dépend du système de fichiers sous-jacent : par exemple, tous les systèmes d'exploitation n'utilisent pas tous le même séparateur entre les éléments d'un chemin.
- Un objet de type Path encapsule le chemin d'un élément du système de fichiers composé d'un ensemble d'éléments organisés de façon hiérarchique grâce à un séparateur spécifique au système.

L'interface Path

- Il existe plusieurs manières de créer un objet de type Path :
  - invoquer la méthode getPath() d'une instance de type FileSystem
  - invoquer la méthode Paths.get() qui invoque la méthode FileSystems.getDefault().getPath()
  - La classe Paths est un helper qui permet de créer facilement des instances de type Path : la méthode get() attend en paramètres un nombre variable d'objets de type String qui sont les éléments du chemin
  - invoquer la méthode toPath() sur un objet de type java.io.File

L'interface Path

- Exemples:
- Path chemin = FileSystems.getDefault().getPath("C:/Users/us/AppData/Local/Temp/monfichier.txt");
- Path chemin1 = Paths.get("C:/Users/us/AppData/Local/Temp/monfichier.txt");

L'interface Path

- Une instance de type Path stocke les éléments de la hiérarchie du chemin sous une forme séquentielle, l'élément le plus haut dans la hiérarchie (après la racine) ayant l'index 0 et l'élément le plus bas ayant l'index n-1, n étant le nombre d'éléments du chemin.
- L'interface Path propose plusieurs méthodes pour retrouver un élément particulier ou un sous-chemin composé de plusieurs éléments en utilisant les index.

L'interface Path

- String getFileName()
- Retourner le nom du dernier élément du chemin. Si le chemin concerne un fichier alors c'est le nom du fichier qui est retourné
- Path getName(int index)
- Retourner l'élément du chemin dont l'index est fourni en paramètre. Le premier élément possède l'index 0
- int getNameCount()
- Retourner le nombre d'éléments du chemin

L'interface Path

- Path getParent()
- Retourner le chemin parent ou null s'il n'existe pas (dans ce cas, le chemin correspond à une racine)
- Path getRoot()
- Retourner la racine d'un chemin absolu (par exemple C:\ sous windows ou / sous Unix) ou null pour un chemin relatif
- String toString()
- Retourner le chemin sous la forme d'une chaîne de caractères
- Path subPath(int beginIndex, int endIndex)
- Retourner un sous-chemin correspondant aux deux index fournis en paramètres

Exemple

```
Path path =
Paths.get("C:/Users/us/AppData/Local/Temp/monfichier.txt");
 System.out.println("toString()     = " + path.toString());
    System.out.println("getFileName()  = " +
path.getFileName());
    System.out.println("getRoot()      = " + path.getRoot());
    System.out.println("getName(0)     = " + path.getName(0));
    System.out.println("getNameCount() = " +
path.getNameCount());
    System.out.println("getParent()    = " + path.getParent());
    System.out.println("subpath(0,3)   = " + path.subpath(0,3));
```

Résultat

toString() = C:\Users\us\AppData\Local\Temp\monfichier.txt

getFileName() = monfichier.txt

getRoot() = C:\

getName(0) = Users

getNameCount() = 6

getParent() = C:\Users\us\AppData\Local\Temp

subpath(0,3) = Users\us\AppData

Exemple

- Une instance de type Path implémente l'interface Iterator qui permet de réaliser une itération sur les éléments du chemin.
- Exemple (code Java 7) :

```
Path path =
Paths.get("C:/Users/us/AppData/Local/Temp/monfichier.txt");
for (Path name : path) {
 System.out.println(name);
}
```

- Résultat :

```
Users
us
AppData
Local
Temp
monfichier.txt
```

La classe Files

- La classe java.nio.file.Files est un helper qui contient plusieurs méthodes statiques permettant de réaliser des opérations sur des fichiers ou des répertoires dont le chemin est encapsulé dans un objet de type Path.
- La classe Files permet de réaliser des opérations de base sur les fichiers et les répertoires : création, ouverture, suppression, test d'existence, changement des permissions, ...
- Ces méthodes concernent notamment :
- La création d'éléments : createDirectory(), createFile(),...
- La manipulation d'éléments : delete(), move(), copy(), ...
- L'obtention du type d'un élément : isRegularFile(), isDirectory(),...
- L'obtention de métadonnées et la gestion des permissions : getAttributes(), isReadable(), isWriteable(), size(), getFileAttributeView(), ...

Les vérifications sur un fichier ou un répertoire

- La classe Files propose deux méthodes pour vérifier l'existence d'un élément dans le système de fichier :
- boolean exists(Path)
- vérifier l'existence sur le système de fichiers de l'élément dont le chemin est encapsulé dans le paramètre de type Path fourni
- boolean notExists(Path)
- vérifier que l'élément dont le chemin est encapsulé dans l'instance de type Path fournie en paramètre n'existe pas sur le système de fichiers

Exemple java 7 (copie)

- Pour copier le fichier test.txt vers un fichier test2.txt :

```
Path source = Paths.get("test.txt");
Path cible = Paths.get("test2.txt");
try {
  Files.copy(source, cible,
StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) { e.printStackTrace();  }
```

- Le troisième argument permet de spécifier les options de copie. Exemples:
  - StandardCopyOption.REPLACE_EXISTING: remplace le fichier cible même s'il existe déjà ;
  - StandardCopyOption.COPY_ATTRIBUTES: copie les attributs du fichier source sur le fichier cible (droits en lecture etc.) ;

Exemple java 7 (déplacement)

- Pour déplacer le fichier test2.txt vers un fichier test3.txt :

```
Path source = Paths.get("test2.txt");
Path cible = Paths.get("test3.txt");
try {
  Files.move(source, cible,
StandardCopyOption.REPLACE_EXISTING);
} catch (IOException e) { e.printStackTrace();  }
```

- On a aussi :
  - une méthode Files.delete(path) qui supprime un fichier ;
  - une méthode Files.createFile(path) qui permet de créer un fichier vide.

La classe Files

- Les méthodes readAllBytes() et readAllLines() permettent de lire l'intégralité du contenu d'un fichier respectivement d'octets et texte. Deux surcharges de la méthode write() permettent d'écrire l'intégralité d'un fichier. Ces méthodes sont à réserver pour de petits fichiers.
- Les méthodes newBufferedReader() et newBufferedWriter() sont des helpers pour faciliter la création d'objets de types BufferedReader et BufferedWriter permettant la lecture et l'écriture de fichiers de type texte en utilisant un tampon.
- Les méthodes newInputStream() et newOutputStream() sont des helpers pour faciliter la création d'objets permettant la lecture et l'écriture de fichiers d'octets.
- Ces quatre méthodes sont des helpers pour créer des objets du package java.io.

La lecture et l'écriture de l'intégralité d'un fichier

- La classe Files propose les méthodes readAllLines() et readAllBytes() qui renvoient respectivement une collection de type List<String> et un tableau d'octets contenant l'intégralité d'un fichier texte ou binaire. L'utilisation de ces méthodes est à réserver pour des fichiers de petites tailles.
- La méthode readAllLines() de la classe Files permet de lire l'intégralité d'un fichier et de renvoyer son contenu sous la forme d'une collection de chaînes de caractères.

```
List<String> lignes =  Files.readAllLines(
  FileSystems.getDefault().getPath("monfichier.txt"),
StandardCharsets.UTF_8);
for (String ligne : lignes)
  System.out.println(ligne);
```

La lecture et l'écriture de l'intégralité d'un fichier

- La méthode readAllLines() attend en paramètre un objet de type Path qui encapsule le chemin du fichier à lire et un objet de type Charset qui précise le jeu d'encodage de caractères du fichier. Elle s'occupe d'ouvrir le fichier, lire le contenu et fermer le flux.
- La méthode readAllBytes() de la classe Files permet de lire l'intégralité d'un fichier et renvoyer son contenu sous la forme d'un tableau d'octets.

```
Path file = FileSystems.getDefault().getPath("monfichier.bin");
byte[] contenu = Files.readAllBytes(file);
```

La lecture et l'écriture de l'intégralité d'un fichier

- La méthode write() permet d'écrire le contenu d'un fichier. Elle possède deux surcharges :
- Path write(Path path, byte[] bytes, OpenOption... options)
- Path write(Path path, Iterable lines, Charset cs, OpenOption... options)

```
Path pathSource = Paths.get("c:/java/source.txt");
Path pathCible = Paths.get("c:/java/cible.txt");
List<String> lignes = Files.readAllLines(pathSource, Charset.defaultCharset());
    Files.write(pathCible, lignes, Charset.defaultCharset());
```

Exemple

```
Path pathSource =
Paths.get("c:/java/source.bin");
Path pathCible = Paths.get("c:/java/cible.bin");
// lire et écrire tout le fichier
byte[] bytes = Files.readAllBytes(pathSource);
Files.write(pathCible, bytes);
```

La lecture et l'écriture bufférisées d'un fichier

- Avant Java 7, pour lire un fichier avec un tampon, il fallait invoquer le constructeur de la classe BufferedReader en lui passant en paramètre un objet de type Reader.
- Exemple :
- BufferedReader in = new BufferedReader(new FileReader("monfichier.txt"));

La lecture et l'écriture bufférisées d'un fichier

- A partir de Java 7, il est possible d'utiliser la méthode newBufferedReader() de la classe Files.
- BufferedReader in = Files.newBufferedReader(Paths.get("monfichier.txt"), Charset.forName("UTF-8"));
- Le résultat est quasiment le même mais il est nécessaire de préciser le jeu d'encodage des caractères. La classe FileReader utilise toujours le jeu par défaut du système. Il est possible d'obtenir ce jeu d'encodage de caractères en invoquant la méthode java.nio.charset.Charset.defaultCharset().
- La méthode newBufferedReader() de la classe Files renvoie un objet de type BufferedReader qui permet de lire le fichier dont le chemin et le jeu de caractères d'encodage sont fournis en paramètres.

Exemple

```
public static void testNewBufferedReader() throws
IOException {
    Path sourcePath = Paths.get("C:/java/temp/monfichier.txt");
    try (BufferedReader reader =
Files.newBufferedReader(sourcePath,
        StandardCharsets.UTF_8)) {
      String line = null;
      while ((line = reader.readLine()) != null) {
        System.out.println(line);
      }
    }
  }
```

Exemple

- La méthode newBufferedReader() ouvre un fichier de type texte pour des lectures avec un tampon. Elle retourne un objet de type BufferedReader.

```
Path fichier = Paths.get("monfichier.txt");
   Charset charset = Charset.forName("US-ASCII");
   try (BufferedReader reader = Files.newBufferedReader(fichier, charset)) {
     String line = null;
     while ((line = reader.readLine()) != null) {
       System.out.println(line);
     }
   } catch (IOException ioe) {
     ioe.printStacktrace();
   }
```

Exemple

- La méthode newBufferedWriter() ouvre un fichier de type texte pour des écritures avec un tampon. Elle retourne un objet de type BufferedWriter.

```
Path fichier = Paths.get("monfichier.txt");
   Charset charset = Charset.forName("US-ASCII");
   String contenu = "Contenu du fichier";
   try (BufferedWriter writer = Files.newBufferedWriter(fichier, charset)) {
     writer.write(contenu, 0, contenu.length());
   } catch (IOException ioe) {
     ioe.printStacktrace();
   }
```

La lecture et l'écriture d'un flux d'octets

- Les méthodes newInputStream() et newOutputStream() permettent d'obtenir une instance de type InputStream et une instance de type OutputStream sur le fichier dont le chemin est fourni en paramètre :

La lecture et l'écriture d'un flux d'octets

- InputStream newInputStream(Path path, OpenOption... options)
- Créer un objet de type InputStream
- OutputStream newOutputStream(Path path, OpenOption... options)
- Créer un objet de type OutputStream

La lecture et l'écriture d'un flux d'octets

- Les méthodes newInputStream() et newOutputStream() attendent en paramètres un objet de type Path et un varargs de type OpenOption.
- La méthode newInputStream() ouvre un fichier pour des lectures sans tampon. Elle retourne un objet de type InputStream.

```
public static void testNewInputStream() throws IOException {
  Path path = Paths.get("c:/java/test/monfichier.txt");
  try (InputStream in = Files.newInputStream(path);
      BufferedReader reader = new BufferedReader(new
InputStreamReader(in))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
     System.out.println(line);
    }
  } catch (IOException x) {
    System.err.println(x);
  }
}
```

La lecture et l'écriture d'un flux d'octets

- La méthode newOutputStream() ouvre un fichier pour des écritures sans tampon. Elle retourne un objet de type OutputStream. Si aucun paramètre de type OpenOption n'est précisé, la méthode va utiliser les paramètres CREATE et TRUNCATE_EXISTING par défaut (créer le fichier s'il n'existe pas et le vider s'il existe).

```
public static void testNewOutputStream() throws IOException {
    Path path = Paths.get("c:/java/test/monfichier.txt");
    try (OutputStream out = Files.newOutputStream(path,
        StandardOpenOption.TRUNCATE_EXISTING,
StandardOpenOption.WRITE)) {
      out.write('X');
    }
  }
```

La gestion des erreurs et la libération des ressources

- La gestion des erreurs et la libération des ressources
- Lors d'opérations d'entrées-sorties de nombreuses erreurs inattendues peuvent survenir, par exemple un fichier qui n'existe pas, un manque de droit d'accès, une erreur de lecture, ...
- Toutes ces erreurs sont encapsulées dans une exception de type IOException ou d'un de ses sous-types. Toutes les méthodes qui réalisent des opérations d'entrées-sorties peuvent lever ces exceptions.
- Avant Java 7, les opérations de type I/O devaient être utilisées dans un bloc try et les exceptions pouvant être levées, traitées dans des blocs catch. La fermeture des flux devait être assurée dans un bloc finally pour garantir son exécution dans tous les cas.

La gestion des erreurs et la libération des ressources

- A partir de Java SE 7, il est préférable d'utiliser l'opérateur try-with-ressources pour assurer la libération automatique des ressources et la gestion des exceptions.
- Exemple

```
Charset charset = Charset.forName("UTF-8");
     String contenu = "Bonjour";
     try (BufferedWriter writer = Files.newBufferedWriter(file,
charset)) {
         writer.write(contenu, 0, contenu.length());
     } catch (IOException ioe) {
         ioe.printStackTrace();
     }
```

JDK 1.7 (gestion des exceptions)

- Contrairement à la gestion de la mémoire (variables, classes, etc.) qui est déléguée au garbage collector (ramasse miette), plusieurs types de ressources doivent être gérées manuellement.
- Les flux sur des fichiers en font parti mais, d'un point de vue plus général, toutes les ressources que doivent être fermées manuellement (les flux réseaux, les connexions à une base de données…).
- Pour ce genre de flux, il faut déclarer une variable en dehors d'un bloc try{…}catch{…} afin qu'elle soit accessible dans les autres blocs d'instructions, le bloc finally par exemple.
- Java 7 initie le « try-with-resources » : déclarer les ressources utilisées directement dans le bloc try(…), ces dernières seront automatiquement fermées à la fin du bloc d'instructions

Exemple copie de fichiers avec Java 7

```
try(FileInputStream fis = new FileInputStream("test.txt");
FileOutputStream fos = new FileOutputStream("test2.txt")) {
  byte[] buf = new byte[8];
  int n = 0;
  while((n = fis.read(buf)) >= 0){
    fos.write(buf);
    for(byte bit : buf)
      System.out.print("\t" + bit + "(" + (char)bit + ")");
       System.out.println("");
  }
  System.out.println("Copie terminée !");
} catch (IOException e) {
  e.printStackTrace();
}
```

La bloc try-with-resources

- Il faut cependant prendre quelques précautions notamment pour ce genre de déclaration :
- try (DataInputStream dis = new DataInputStream(new FileInputStream("test.txt"))) {

```
//…
}
```

- Le fait d'avoir des ressources encapsulées dans d'autres ne rend pas « visible » les ressources encapsulées. Dans le cas précédent, si une exception est levée, le flux correspondant à l'objet FileInputStream ne sera pas fermé. Pour pallier ce problème il suffit de bien découper toutes les ressources à utiliser, comme ceci :

La bloc try-with-resources

```
try (FileInputStream fis = new FileInputStream("test.txt");
        DataInputStream dis = new DataInputStream(fis)) {
        //…
        }
```

- Pour rendre la fermeture automatique possible, les développeurs de la plateforme Java 7 ont créé une nouvelle interface :java.lang.AutoCloseable. Seuls les objets implémentant cette interface peuvent être utilisés de la sorte (voir la liste des classes autorisées dans la documentation java)
- la classe File n'en fait pas parti

L'interopérabilité avec le code existant

- Les objets de type Path obtenus sur le système de fichiers par défaut sont interopérables avec des objets de type java.io.File.
- Pour faciliter la portabilité du code utilisant l'API java.io vers NIO2, la classe java.io.File propose la méthode toPath() qui crée une instance de type Path à partir des informations encapsulées dans l'instance de type File.
- Path input = file.toPath();

L'interopérabilité avec le code existant

- Il est ainsi facile de bénéficier des fonctionnalités offertes par NIO2 sans avoir à tout réécrire.
- Exemple :

```
file.delete();
```

- Il est possible de réécrire cette portion de code en utilisant NIO2.
- Exemple

```
Path fp = file.toPath();
Files.delete(fp);
```

L'interopérabilité avec le code existant

- Inversement, la classe Path propose la méthode toFile() permettant de créer une instance de la classe java.io.File qui correspond aux informations encapsulées dans l'instance de type Path.

Résumé : Java 7

Création **java.nio.file.Files**
– Flux binaire en lecture
InputStream **newInputStream(Path)**
– Flux binaire en écriture
OutputStream **newOutputStream(Path)**
– Flux textuel en lecture
BufferedReader **newBufferedReader(Path, Charset)**
BufferedReader **newBufferedReader(Path)**
– Encodage UTF-8 par défaut
– Flux textuel en écriture
BufferedWriter **newBufferedWriter(Path)**
BufferedWriter **newBufferedWriter(Path, Charset)**
– Encodage UTF-8 par défaut

Exemple java 7

```
Path path = Paths.get("test.txt");
System.out.println("Chemin absolu du fichier : " +
path.toAbsolutePath());
System.out.println("Est-ce qu'il existe ? " +
Files.exists(path));
System.out.println("Nom du fichier : " +
path.getFileName());
System.out.println("Est-ce un répertoire ? " +
Files.isDirectory(path));
```

La classe Scanner

- Version 5 de Java : Un Scanner peut se brancher sur à peu près n'importe quelle source : InputStream, Reader, File... et bien sûr une simple String.
- un objet de type Scanner peut prendre un paramètre qui correspond à l'entrée standard en Java.
- public **Scanner**(File *source*) throws FileNotFoundException
- public **Scanner**(File *source*, String *charsetName*) throws FileNotFoundException
- public **Scanner**(InputStream *source*)
- public **Scanner**(InputStream *source*, String *charsetName*)
- public **Scanner**(String *source*)