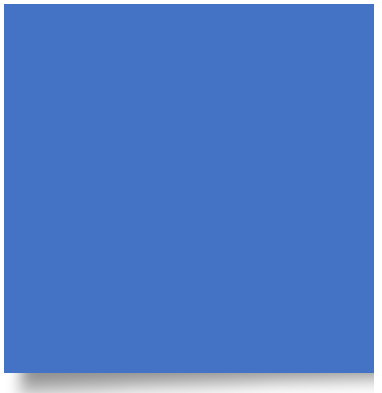
PROGRAMMATION EN JAVA

Semaine 7: Java IO



Enseignant: NGUYỄN Thị Minh Tuyền

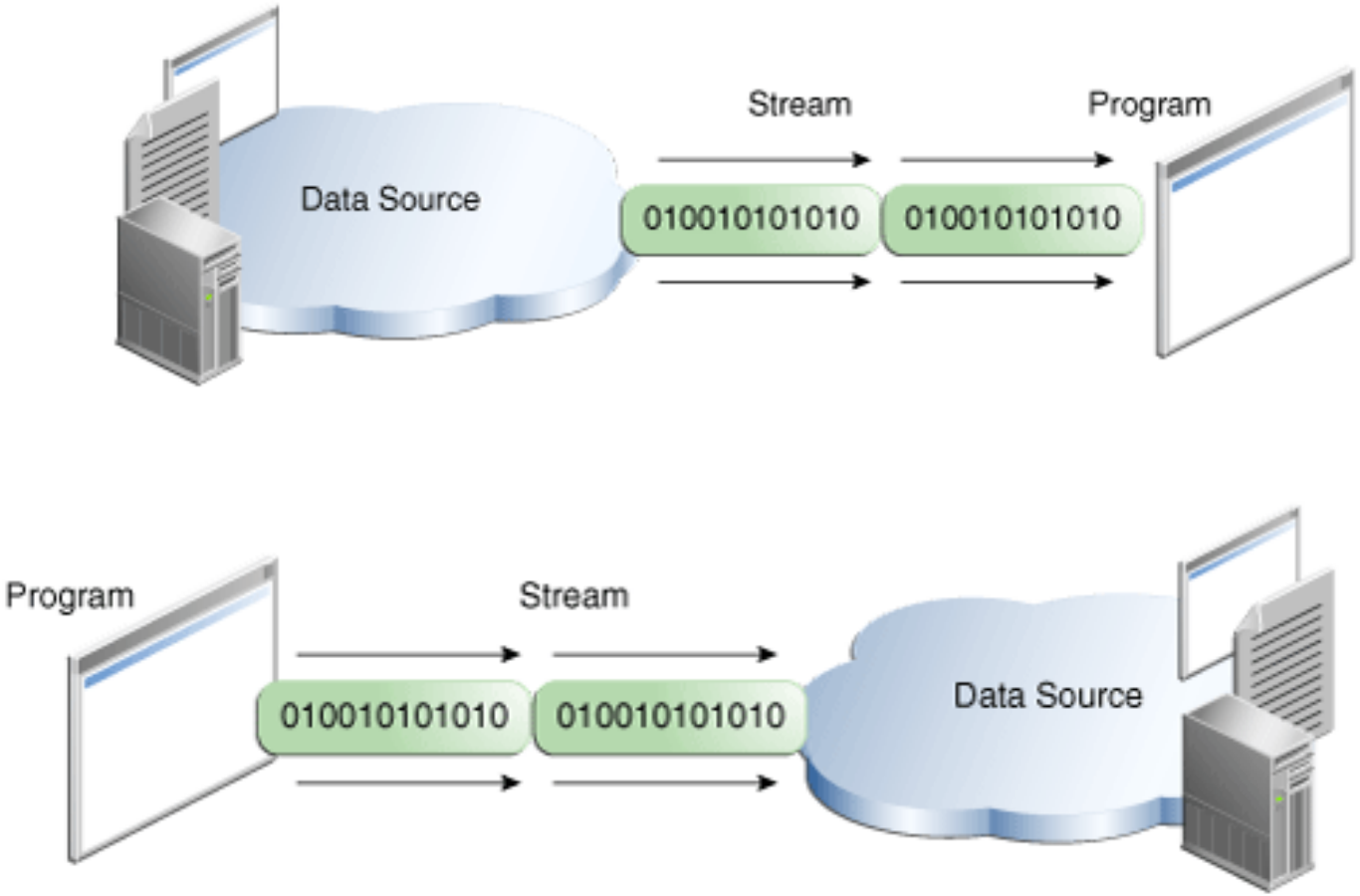
Plan du cours

1. **Java I/O sont basées sur des flux**
2. **Classes de flux d'octets**
3. **Lecture et écriture de données binaires**
4. **Classes de flux de caractères**

Plan du cours

1. **Java I/O sont basées sur des flux**
2. Classes de flux d'octets
3. Lecture et écriture de données binaires
4. Classes de flux de caractères

Flux


Stream
Program
Data Source
010010101010
010010101010
Program
Stream
010010101010
010010101010
Data Source

Java IO sont basées sur des flux

- Les programmes Java effectuent IO via des flux.
- Un flux d'IO est une abstraction qui produit ou consomme des informations.
- Un flux est lié à un périphérique physique par le système de Java IO.
- Tous les flux se comportent de la même manière, même si les périphériques physiques réels auxquels ils sont liés diffèrent - les mêmes classes et méthodes d'IO peuvent être appliquées à différents types de périphériques.
- Java implémente les flux d'IO dans les hiérarchies de classes définies dans le package `java.io`.

Classes de gestion des flux

	Flux d'octets	Flux de caractères
Flux d'entrée	`InputStream`	`Reader`
Flux de sortie	`OutputStream`	`Writer`

- `Reader`
  - flux en lecture sur des ensembles de caractères
- `Writer`
  - flux en écriture sur des ensembles de caractères
- `InputStream`
  - flux en lecture sur des ensembles d'octets
- `OutputStream`
  - flux en écriture sur des ensembles d'octets

Flux d'octets et flux de caractères

- Deux types de flux d'IO: octet et caractère.
- Les flux d'octets constituent un moyen pratique pour gérer l'entrée et la sortie d'octets.
  - Peut être utilisé lors de la lecture ou de l'écriture de données binaires.
  - Sont particulièrement utiles lorsque vous travaillez avec des fichiers.
- Les flux de caractères sont conçus pour gérer l'entrée et la sortie des caractères.
  - Ils utilisent Unicode → peuvent être internationalisés.
  - Dans certains cas, les flux de caractères sont plus efficaces que les flux d'octets.
- **Remarque**:
  - Au niveau le plus bas, toutes les I/O sont toujours orientées octets.
  - Les flux basés sur des caractères fournissent simplement un moyen pratique et efficace pour gérer les caractères.

Classes de flux d'octets

Byte Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that allows bytes to be returned to the stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Classes de flux de caractères

Character Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Flux prédéfinis [1]

- La classe System, défini dans `java.lang`, encapsule plusieurs aspects de l'environnement d'exécution.
  - Il contient trois variables de flux prédéfinies: `in`, `out` et `err`.
  - Ces champs sont déclarés `public`, `final` et `static` → peuvent être utilisés par n'importe quelle autre partie de votre programme et sans référence à un objet système spécifique.
- `System.out` fait référence au flux de sortie standard.
  - Défaut: console.
- `System.in` fait référence à l'entrée standard.
  - Défaut: clavier.
- `System.err` fait référence au flux d'erreur standard.
  - Défaut: console.

Flux prédéfinis [2]

- Ces flux peuvent être redirigés vers n'importe quel périphérique d'E/S compatible.
- `System.in` est un objet de type `InputStream`; `System.out` et `System.err` sont des objets de type `PrintStream`.
  - Il s'agit de flux d'octets, même s'ils sont généralement utilisés pour lire et écrire des caractères depuis et vers la console.
  - Raison: les flux prédéfinis faisaient partie de la spécification d'origine pour Java, qui n'incluait pas les flux de caractères.

Méthodes définies par `InputStream`

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Subsequent read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns true if **mark( )**/**reset( )** are supported by the invoking stream.
static InputStream nullInputStream( )	Returns an open, but null stream, which is a stream that contains no data. Thus, the stream is always at the end of the stream and no input can be obtained. The stream can, however, be closed. (Added by JDK 11.)
int read( )	Returns an integer representation of the next available byte of input. −1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. −1 is returned when an attempt is made to read at the end of the stream.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. −1 is returned when an attempt is made to read at the end of the stream.
byte[ ] readAllBytes( )	Reads and returns, in the form of an array of bytes, all bytes available in the stream. An attempt to read at the end of the stream results in an empty array.
byte[ ] readNBytes(int *numBytes*)	Attempts to read *numBytes* bytes, returning the result in a byte array. If the end of the stream is reached before *numBytes* bytes have been read, then the returned array will contain less than *numBytes* bytes. (Added by JDK 11.)
int readNBytes(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. An attempt to read at the end of the stream results in zero bytes being read.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.
long transferTo(OutputStream *outStrm*)	Copies the contents of the invoking stream to *outStrm*, returning the number of bytes copied.

Méthodes définies par `OutputStream`

Method	Description
void close( )	Closes the output stream. Subsequent write attempts will generate an **IOException**.
void flush( )	Causes any output that has been buffered to be sent to its destination. That is, it flushes the output buffer.
static OutputStream nullOutputStream( )	Returns an open, but null output stream, which is a stream to which no output is written. The stream can, however, be closed. (Added by JDK 11.)
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

Entrée de la console de lecture

- Peut utiliser des flux d'octets ou de caractères.
- Méthode préférée: flux orienté caractère.
  - Rendez votre programme plus facile à internationaliser et plus facile à maintenir.
  - Il est également plus pratique d'opérer directement sur les caractères plutôt que de faire des allers-retours entre les caractères et les octets.
- `System.in` est une instance de InputStream → avoir automatiquement accès aux méthodes définies par `InputStream` → nous pouvons utiliser la méthode `read()` pour lire les octets de `System.in`.
- Trois versions de read ():

```
int read() throws IOException
int read(byte data[]) throws IOException
int read(byte data[], int off, int len)
                                throws IOException
```

Exemple


```
1.  import java.io.*;
2.  class ReadBytes {
3.      public static void main(String args[])
4.                                  throws IOException {
5.          byte data[] = new byte[10];
6.
7.          System.out.println("Enter some characters.");
8.          System.in.read(data);
9.          System.out.print("You entered: ");
10.         for(byte c:data)
11.             System.out.print((char)c);
12.         System.out.println();
13.     }
14. }
```

Écriture de la sortie de la console [1]

- Pour le code le plus portable, les flux de caractères sont recommandés.
- Parce que `System.out` est un flux d'octets, la sortie de la console basée sur les octets est encore largement utilisée.
- La sortie de la console est plus facile à réaliser avec `print()` et `println()`.
  - Ces méthodes sont définies par la classe `PrintStream`
  - Même si `System.out` est un flux d'octets, il est toujours acceptable d'utiliser ce flux pour une sortie console simple.
- `PrintStream` est un flux de sortie dérivé de `OutputStream` - il implémente également la méthode de bas niveau `write()` - possible d'écrire sur la console en utilisant `write()`.

Écriture de la sortie de la console [2]

- La forme la plus simple de `write()` définie par `PrintStream`:

```
void write (int byteval)
```

- Cette méthode écrit l'octet spécifié par `byteval` dans le fichier.
- Bien que `byteval` soit déclaré comme un entier, seuls les 8 bits de poids faible sont écrits.

Exemple


```
1.  // Demonstrate System.out.write().
2.  class WriteDemo {
3.      public static void main(String args[]) {
4.          int b;
5.
6.          b = 'X';
7.          System.out.write(b);
8.          System.out.write('\n');
9.      }
10. }
```

Lecture et écriture de fichiers à l'aide de flux d'octets

- Tous les fichiers sont orientés octets, et Java fournit des méthodes pour lire et écrire des octets depuis et vers un fichier → la lecture et l'écriture de fichiers à l'aide de flux d'octets sont très courantes.
- Java vous permet également d'encapsuler un flux de fichiers orienté octets dans un objet basé sur des caractères.
- Pour créer un flux d'octets lié à un fichier, utilisez `FileInputStream` ou `FileOutputStream`.
- Pour ouvrir un fichier:
  - créer un objet d'une de ces classes, en spécifiant le nom du fichier comme argument pour le constructeur.
  - Une fois le fichier ouvert, vous pouvez y lire ou y écrire.

Saisie à partir d'un fichier

- Un fichier est ouvert pour entrée en créant un objet `FileInputStream`.

```
FileInputStream(String fileName)
        throws FileNotFoundException
```

  - `fileName` spécifie le nom du fichier à ouvrir.
  - Si le fichier n'existe pas, `FileNotFoundException` est levée.
  - `FileNotFoundException` est une sous-classe d'`IOException`.
- Pour lire à partir d'un fichier:

```
int read() throws IOException
```

- Lorsque vous avez terminé avec un fichier, vous devez le fermer en appelant `close()`:

```
void close() throws IOException
```

Exemple [1]


```
1.  import java.io.*;
2.  class ShowFile {
3.      public static void main(String args[]) {
4.      int i; FileInputStream fin;
5.      // First make sure that a file has been specified.
6.      if (args.length != 1) {
7.          System.out.println("Usage: ShowFile File");
8.          return;
9.      }
10.     try {
11.         fin = new FileInputStream(args[0]);
12.     } catch (FileNotFoundException exc) {
13.         System.out.println("File Not Found"); return;
14.     }
15.     // ...
```

Exemple [2]


```
1.      // ...
2.      try { // read bytes until EOF is encountered
3.          do {
4.              i = fin.read();
5.              if (i != -1)
6.                  System.out.print((char) i);
7.          } while (i != -1);
8.      } catch (IOException exc) {
9.          System.out.println("Error reading file.");
10.     }
11.     try {
12.         fin.close();
13.     } catch (IOException exc) {
14.         System.out.println("Error closing file.");
15.     }
16. }
```

Version 2


```
1.  import java.io.*;
2.  class ShowFile1 {
3.     public static void main(String args[]) {
4.        int i;
5.        FileInputStream fin = null;
6.
7.        // First, confirm that a file name has been specified.
8.        if (args.length != 1) {
9.              System.out.println("Usage: ShowFile filename");
10.             return;
11.       }
12.       /* The following code opens a file, reads characters
13.          until EOF is encountered, and then closes the file
14.          via a finally block. */
```

```
1.          try {
2.              fin = new FileInputStream(args[0]);
3.              do {
4.                  i = fin.read();
5.                  if (i != -1) System.out.print((char) i);
6.              } while (i != -1);
7.          } catch (FileNotFoundException exc) {
8.                  System.out.println("File Not Found.");
9.          } catch (IOException exc) {
10.                 System.out.println("An I/O Error Occurred");
11.         } finally { // Close file in all cases.
12.             try {
13.                 if (fin != null) fin.close();
14.             } catch (IOException exc) {
15.                 System.out.println("Error Closing File");
16.             }
17.         }
18.     }
19. }
```

Écrire dans un fichier [1]

- Pour ouvrir un fichier pour la sortie, créez un objet `FileOutputStream`.

```
FileOutputStream(String fileName)
                throws FileNotFoundException
FileOutputStream(String fileName, boolean append)
                throws FileNotFoundException
```

- Si le fichier n'est pas créé: `FileNotFoundException` est levée.
- Dans la première syntaxe: lorsqu'un fichier de sortie est ouvert, tout fichier préexistant du même nom est détruit.
- Dans la deuxième syntaxe: si append est vrai, la sortie est ajoutée à la fin du fichier.

Écrire dans un fichier [2]

- Pour écrire dans un fichier: utilisez `write()`

```
void write(int byteval) throws IOException
```

  - Écrit l'octet spécifié par `byteval` dans le fichier.
  - Bien que l'octet soit déclaré comme un entier, seuls les 8 bits de poids faible sont écrits dans le fichier.
  - Si une erreur se produit pendant l'écriture, une exception `IOException` est levée.
- Une fois l'écriture au un fichier de sortie est terminé: il fau le fermer en utilisant `close()`

```
void close() throws IOException
```

  - La fermeture d'un fichier libère les ressources système allouées au fichier, leur permettant d'être utilisées par un autre fichier.
  - Cela permet également de garantir que toute sortie restante dans un tampon de sortie est réellement écrite sur le périphérique physique.

```
1. import java.io.*;
2. class CopyFile {
3.    public static void main(String args[])
4.                                throws IOException {
5.       int i;
6.       FileInputStream fin = null;
7.       FileOutputStream fout = null;
8.       // First, make sure that both files has been specified.
9.       if (args.length != 2) {
10.          System.out.println("Usage: CopyFile from to");
11.          return;
12.      }
13.      // Copy a File.
14.      try { // Attempt to open the files.
15.          fin = new FileInputStream(args[0]);
16.          fout = new FileOutputStream(args[1]);
17.          // ...
```

```
1.              do {
2.                  i = fin.read();
3.                  if (i != -1) fout.write(i);
4.              } while (i != -1);
5.          } catch (IOException exc) {
6.                  System.out.println("I/O Error: " + exc);
7.          } finally {
8.              try {
9.                  if (fin != null) fin.close();
10.             } catch (IOException exc) {
11.                 System.out.println("Error Closing Input File");
12.             }
13.             try { if (fout != null) fout.close();
14.             } catch (IOException exc) {
15.                 System.out.println("Error Closing Output File");
16.             }
17.         }
18.     }
19. }
```

Fermeture automatique d'un fichier

- À partir de JDK 7: Java offre une autre façon plus rationalisée de gérer les ressources, telles que les flux de fichiers, en automatisant le processus de fermeture.
  - Il est basé sur une autre version de l'instruction try appelée `trywith-resources` (gestion automatique des ressources).
  - Avantage: empêche les situations dans lesquelles un fichier (ou une autre ressource) n'est pas libéré par inadvertance après qu'il n'est plus nécessaire.
  - Oublier de fermer un fichier peut entraîner des fuites de mémoire et entraîner d'autres problèmes.

```
try (resources-specification) {
    // use the resource
}
```

- `resource-specification` est une instruction qui déclare et initialise une ressource, telle qu'un fichier.

```
1. import java.io.*;
2. class ShowFile2 {
3.     public static void main(String args[]) {
4.         int i;
5.         //First, make sure that a file name has been specified.
6.         if (args.length != 1) {
7.             System.out.println("Usage: ShowFile filename");
8.             return;
9.         }
10.        /* Use try-with resources to open a file and then
11.           automatically close it when the try block is left. */
12.        try(FileInputStream fin = new FileInputStream(args[0])){
13.            do {
14.                i = fin.read();
15.                if (i != -1) System.out.print((char) i);
16.            } while (i != -1);
17.        } catch (IOException exc) {
18.                System.out.println("I/O Error: " + exc);
19.        }
20.    }
21. }
```

```
1.  import java.io.*;
2.  class CopyFile1 {
3.      public static void main(String args[])
4.                                  throws IOException {
5.          int i;
6.          // First, confirm that both files has been specified.
7.          if (args.length != 2) {
8.                  System.out.println("Usage: CopyFile from to");
9.                  return;
10.         }
11.         // Open and manage two files via the try statement.
12.         try(FileInputStream fin=new FileInputStream(args[0]);
13.           FileOutputStream fout=new FileOutputStream(args[1])){
14.            do {
15.                i = fin.read(); if (i != -1) fout.write(i);
16.            } while (i != -1);
17.         } catch (IOException exc) {
18.                 System.out.println("I/O Error: " + exc);
19.         }
20.     }
21. }
```

Lecture et écriture de données binaires

- Utilisez `DataInputStream` et `DataOutputStream`.

DataOutputStream

- `DataOutputStream` implémente l'interface de `DataOutput`.
  - Cette interface définit des méthodes qui écrivent tous les types primitifs de Java dans un fichier.
  - Ces données sont écrites en utilisant son format binaire interne et non sa forme textuelle lisible par l'homme.
- Constructeur

```
DataOutputStream(OutputStream outputStream)
```

  - `outputStream` est le flux dans lequel les données sont écrites.
  - Pour écrire la sortie dans un fichier, vous pouvez utiliser l'objet créé par `FileOutputStream` pour ce paramètre.

Méthodes couramment utilisées définies par `DataOutputStream`

Output Method	Purpose
void writeBoolean(boolean *val*)	Writes the **boolean** specified by *val*.
void writeByte(int *val*)	Writes the low-order byte specified by *val*.
void writeChar(int *val*)	Writes the value specified by *val* as a **char**.
void writeDouble(double *val*)	Writes the **double** specified by *val*.
void writeFloat(float *val*)	Writes the **float** specified by *val*.
void writeInt(int *val*)	Writes the **int** specified by *val*.
void writeLong(long *val*)	Writes the **long** specified by *val*.
void writeShort(int *val*)	Writes the value specified by *val* as a **short**.

Chaque méthode lève une exception `IOException` en cas d'échec.

DataInputStream

- `DataInputStream` implémente l'interface de `DataInput`, qui fournit des méthodes pour lire tous les types primitifs de Java.
- `DataInputStream` utilise une instance `InputStream` comme base, en la superposant avec des méthodes qui lisent les différents types de données Java.
- `DataInputStream` lit les données dans son format binaire, pas dans sa forme lisible par l'homme.
- Constructeur:

```
DataInputStream(InputStream inputStream)
```

  - `inputStream` est le flux qui est lié à l'instance de `DataInputStream` en cours de création.

Méthodes d'entrée couramment utilisées définies par DataInputStream

Input Method	Purpose
boolean readBoolean( )	Reads a **boolean**.
byte readByte( )	Reads a **byte**.
char readChar( )	Reads a **char**.
double readDouble( )	Reads a **double**.
float readFloat( )	Reads a **float**.
int readInt( )	Reads an **int**.
long readLong( )	Reads a **long**.
short readShort( )	Reads a **short**.

Chaque méthode lève une exception `IOException` en cas d'échec.

```
1.  import java.io.*;
2.  class RWData {
3.      public static void main(String args[]) {
4.          int i = 10; double d = 1023.56; boolean b = true;
5.          // Write some values.
6.          try (DataOutputStream dataOut = new
7.            DataOutputStream(new FileOutputStream("testdata"))){
8.              System.out.println("Writing " + i);
9.              dataOut.writeInt(i);
10.             System.out.println("Writing " + d);
11.             dataOut.writeDouble(d);
12.             System.out.println("Writing " + b);
13.             dataOut.writeBoolean(b);
14.             System.out.println("Writing " + 12.2 * 7.4);
15.             dataOut.writeDouble(12.2 * 7.4);
16.         } catch (IOException exc) {
17.             System.out.println("Write error.");
18.             return;
19.         }
20.         System.out.println();
```

```
1.          // ...
2.          // Now, read them back.
3.          try (DataInputStream dataIn = new
4.            DataInputStream(new FileInputStream("testdata"))) {
5.              i = dataIn.readInt();
6.              System.out.println("Reading " + i);
7.              d = dataIn.readDouble();
8.              System.out.println("Reading " + d);
9.              b = dataIn.readBoolean();
10.             System.out.println("Reading " + b);
11.             d = dataIn.readDouble();
12.             System.out.println("Reading " + d);
13.         } catch (IOException exc) {
14.             System.out.println("Read error.");
15.         }
16.     }
17. }
```

Exercice 1: Comparer deux fichiers

Version 1:

- Écrivez un programme qui vérifie si deux fichiers ont le même contenu.

Exercice 2: Comparer deux fichiers

Version 2:

- Vérifiez si nous comparons le même fichier, affichez le message et sortez le programme.
- Écrivez un programme qui vérifie si deux fichiers ont le même contenu (en ignorant la casse des lettres).
- Si deux fichiers diffèrent, le programme affichera la position où les fichiers diffèrent.

Accès direct à un fichier binaire [1]

- Pour accéder au contenu d'un fichier dans un ordre aléatoire → utilisez `RandomAccessFile`.
- `RandomAccessFile` n'est pas dérivé de `InputStream` ou `OutputStream`.
  - Il implémente les interfaces `DataInput` et `DataOutput`, qui définissent les méthodes d'E/S de base.
  - Il prend également en charge les demandes de positionnement, c-t-d, vous pouvez positionner le pointeur de fichier dans le fichier.
- Constructeur:

```
RandomAccessFile(String fileName, String access)
                                    throws FileNotFoundException
```

  - `fileName` : le nom du fichier est transmis dans
  - `access`: le type d'accès au fichier autorisé(”r”: read, ”w”: write, “rw”:read-write)

Accès direct à un fichier binaire [2]

- Pour définir la position actuelle du pointeur de fichier dans le fichier:

  `void seek(long newPos) throws IOException`

  - `newPos`: la nouvelle position, en octets, du pointeur de fichier depuis le début du fichier.
  - Après un appel à `seek()`, la prochaine opération de lecture ou d'écriture se produira à la nouvelle position du fichier.
- Car `RandomAccessFile` implémente les interfaces `DataInput` et `DataOuput`, des méthodes pour lire et écrire les types primitifs sont disponibles. Les méthodes `read()` et `write()` sont également prises en charge.

```
1.  import java.io.*;
2.  class RandomAccessDemo {
3.      public static void main(String args[]) {
4.          double data[] ={ 19.4, 10.1, 123.54, 33.0, 87.9, 74.25};
5.          double d;
6.          // Open and use a random access file.
7.          try (RandomAccessFile raf = new
8.                          RandomAccessFile("random.dat", "rw")) {
9.              // Write values to the file.
10.             for (int i = 0; i < data.length; i++) {
11.                 raf.writeDouble(data[i]);
12.             }
13.             // Now, read back specific values
14.             raf.seek(0); // seek to first double
15.             d = raf.readDouble();
16.             System.out.println("First value is " + d);
17.             raf.seek(8); // seek to second double
18.             d = raf.readDouble();
19.             System.out.println("Second value is " + d);
```

```
1.              // ...
2.              raf.seek(8 * 3); // seek to fourth double
3.              d = raf.readDouble();
4.              System.out.println("Fourth value is " + d);
5.
6.              System.out.println();
7.
8.              // Now, read every other value.
9.              System.out.println("Here is every other value: ");
10.             for (int i = 0; i < data.length; i += 2) {
11.                 raf.seek(8 * i); // seek to ith double
12.                 d = raf.readDouble();
13.                 System.out.print(d + " ");
14.             }
15.         } catch (IOException exc) {
16.                 System.out.println("I/O Error: " + exc);
17.         }
18.     }
19. }
```

Utilisation des flux basés sur les caractères de Java

- Les flux d'octets de Java sont à la fois puissants et flexibles, mais ceci n'est pas un moyen idéal pour gérer les I/O basées sur les caractères → classes de flux de caractères.
- Au sommet de la hiérarchie des flux de caractères se trouvent les classes abstraites `Reader` et `Writer`.
- La plupart des méthodes peuvent déclencher une `IOException` en cas d'erreur.
- Les méthodes définies par ces deux classes abstraites sont disponibles pour toutes leurs sous-classes, forment un ensemble minimal de fonctions d'I/O que tous les flux de caractères auront.

Method	Description
abstract void close( )	Closes the input source. Subsequent read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )**/**reset( )** are supported on this stream.
static Reader nullReader( )	Returns an open, but null reader, which is a reader that contains no data. Thus, the reader is always at the end of the stream and no input can be obtained. The reader can, however, be closed. (Added by JDK 11.)
int read( )	Returns an integer representation of the next available character from the invoking input stream. −1 is returned when an attempt is made to read at the end of the stream.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. −1 is returned when an attempt is made to read at the end of the stream.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. −1 is returned when an attempt is made to read at the end of the stream.
int read(CharBuffer *buffer*)	Attempts to fill the buffer specified by *buffer*, returning the number of characters successfully read. −1 is returned when an attempt is made to read at the end of the stream. **CharBuffer** is a class that encapsulates a sequence of characters, such as a string.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.
long transferTo(Writer *writer*)	Copies the contents of the invoking reader to *writer*, returning the number of characters copied. (Added by JDK 10.)

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*)	Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream. **CharSequence** is an interface that defines read-only operations on a sequence of characters.
Writer append(CharSequence *chars*, int *begin*, int *end*)	Appends the sequence of *chars* starting at *begin* and stopping with *end* to the end of the invoking output stream. Returns a reference to the invoking stream. **CharSequence** is an interface that defines read-only operations on a sequence of characters.
abstract void close( )	Closes the output stream. Subsequent write attempts will generate an **IOException**.
abstract void flush( )	Causes any output that has been buffered to be sent to its destination. That is, it flushes the output buffer.
static Writer nullWriter( )	Returns an open, but null output writer, which is a writer to which no output is written. The writer can, however, be closed. (Added by JDK 11.)
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **char**.
void write(char *buffer*[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *str*, beginning at the specified *offset*.

Entrée de console à l'aide de flux de caractères

- Utilisez `InputStreamReader`, qui convertit les octets en caractères. Pour obtenir un objet `InputStreamReader` lié à `System.in`, utilisez le constructeur :

```
InputStreamReader(InputStream inputStream)
```

  - Car `System.in` fait référence à un objet de type `InputStream`, il peut être utilisé pour `inputStream`.

- Ensuite, utilisez l'objet produit par `InputStreamReader`, construisez un `BufferedReader` en utilisant le constructeur :

```
BufferedReader(Reader inputReader)
```

- `inputReader` est le flux qui est lié à l'instance de `BufferedReader` en cours de création.
- En mettant tout cela ensemble, nous avons :

```
BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));
```

Lecture de caractères

- Utilisez la méthode `read()` définie par `BufferedReader` de la même manière qu'ils ont été lus à l'aide de flux d'octets.

```
int read() throws IOException
int read(char data[]) throws IOException
int read(char data[], int start, int max)
                                    throws IOException
```

Exemple


```
1.  import java.io.*;
2.  class ReadChars {
3.     public static void main(String args[]) throws IOException{
4.       char c;
5.       BufferedReader br = new BufferedReader(new
6.                                   InputStreamReader(System.in));
7.       System.out.println("Enter characters, period to quit.");
8.       // read characters
9.       do {
10.           c = (char) br.read();
11.           System.out.println(c);
12.      } while (c != '.');
13.   }
14. }
```

Lecture des Strings


```
String readLine() throws IOException
```

- Il renvoie un objet `String` qui contient les caractères lus. Il renvoie `null` si une tentative de lecture est effectuée à la fin du flux.

Exemple


```
1.  import java.io.*;
2.  class ReadLines {
3.     public static void main(String args[]) throws IOException{
4.        // create a BufferedReader using System.in
5.        BufferedReader br =
6.          new BufferedReader(new InputStreamReader(System.in));
7.        String str;
8.
9.        System.out.println("Enter lines of text.");
10.       System.out.println("Enter 'stop' to quit.");
11.       do {
12.          str = br.readLine();
13.          System.out.println(str);
14.       } while (!str.equals("stop"));
15.    }
16. }
```

Sortie de console à l'aide de flux de caractères

- La méthode préférée d'écriture sur la console lors de l'utilisation de Java: utiliser `PrintWriter` qui est l'une des classes basées sur les caractères.
- Constructeur:

```
PrintWriter(OutputStream outputStream,
            boolean flushingOn)
```

  - `outputStream` est un objet de type `OutputStream` et `flushingOn` contrôle si Java vide le flux de sortie chaque fois qu'une méthode `println()` (entre autres) est appelée.
  - Si `flushingOn` est `true`, le rinçage a lieu automatiquement.
  - Si `false`, e rinçage n'est pas automatique.
- `PrintWriter` prend en charge les méthodes `print()` et `println()` pour tous les types, y compris `Object`.

Exemple


```
1.  //Demonstrate PrintWriter.
2.  import java.io.*;
3.  public class PrintWriterDemo {
4.      public static void main(String args[]) {
5.          PrintWriter pw = new PrintWriter(System.out, true);
6.          int i = 10;
7.          double d = 123.65;
8.
9.          pw.println("Using a PrintWriter.");
10.         pw.println(i);
11.         pw.println(d);
12.         pw.println(i + " + " + d + " is " + (i + d));
13.     }
14. }
```

File I/O à l'aide de flux de caractères

- Utilisez les classes `FileReader` et `FileWriter`

FileWriter

- `FileWriter` crée un `Writer` que vous pouvez utiliser pour écrire dans un fichier. Deux constructeurs couramment utilisés:

```
FileWriter(String fileName) throws IOException
FileWriter(String fileName, boolean append)
                                    throws IOException
```

  - `fileName` est le nom de chemin complet d'un fichier.
  - Si `append` est `true`, la sortie est ajoutée à la fin du fichier. Sinon, le fichier est écrasé.
  - `FileWriter` est dérivé de `OutputStreamWriter` et `Writer`

```
1.  import java.io.*;
2.  class KtoD {
3.      public static void main(String args[]) {
4.          String str;
5.          BufferedReader br = new BufferedReader(new
6.                                  InputStreamReader(System.in));
7.          System.out.println("Enter text ('stop' to quit).");
8.          try (FileWriter fw = new FileWriter("test.txt")) {
9.              do {
10.                 System.out.print(": ");
11.                 str = br.readLine();
12.                 if (str.compareTo("stop") == 0) break;
13.                 str = str + "\r\n"; // add newline
14.                 fw.write(str);
15.             } while (str.compareTo("stop") != 0);
16.         } catch (IOException exc) {
17.             System.out.println("I/O Error: " + exc);
18.         }
19.     }
20. }
```

FileReader

- La classe `FileReader` crée un `Reader` que vous pouvez utiliser pour lire le contenu d'un fichier. Un constructeur couramment utilisé::

```
FileReader(String fileName)
                throws FileNotFoundException
```

  - `fileName` est le nom de chemin complet d'un fichier.
  - Il lève une exception `FileNotFoundException` si le fichier n'existe pas.
  - `FileReader` est dérivé de `InputStreamReader` et `Reader`.

```
1.  import java.io.*;
2.  class DtoS {
3.      public static void main(String args[]) {
4.          String s;
5.          /* Create and use a FileReader wrapped in a
6.                                                  BufferedReader. */
7.          try (BufferedReader br = new BufferedReader(
8.                                      new FileReader("test.txt"))) {
9.              while ((s = br.readLine()) != null) {
10.                 System.out.println(s);
11.             }
12.         } catch (IOException exc) {
13.             System.out.println("I/O Error: " + exc);
14.         }
15.     }
16. }
```

Utilisation de wrappers de types Java pour convertir des chaînes numériques

- La méthode `println()` de Java fournit un moyen pratique de sortir différents types de données vers la console → `println()` convertit automatiquement les valeurs numériques en leur forme lisible par l'homme.
- Cependant, des méthodes comme `read()` ne fournissent pas de fonctionnalité parallèle qui lit et convertit une chaîne contenant une valeur numérique dans son format binaire interne.
- Java fournit d'autres façons: les wrappers de type de Java.
- Les wrappers de type sont `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character`, and `Boolean`.
  - Offre un large éventail de méthodes qui vous permettent d'intégrer pleinement les types primitifs dans la hiérarchie d'objets Java.

Méthodes de conversion

Wrapper	Conversion Method
Double	static double parseDouble(String *str*) throws NumberFormatException
Float	static float parseFloat(String *str*) throws NumberFormatException
Long	static long parseLong(String *str*) throws NumberFormatException
Integer	static int parseInt(String *str*) throws NumberFormatException
Short	static short parseShort(String *str*) throws NumberFormatException
Byte	static byte parseByte(String *str*) throws NumberFormatException

```
1.  import java.io.*;
2.  class AvgNums {
3.      public static void main(String args[])
4.                                  throws IOException{
5.          // create a BufferedReader using System.in
6.          BufferedReader br = new BufferedReader(new
7.                                  InputStreamReader(System.in));
8.          String str; int n; double sum = 0.0; double avg, t;
9.          System.out.print("How many numbers will you enter: ");
10.         str = br.readLine();
11.         try {
12.                 n = Integer.parseInt(str);
13.         } catch (NumberFormatException exc) {
14.                 System.out.println("Invalid format");
15.                 n = 0;
16.         }
17.         // ...
```

```
1.          // ...
2.          System.out.println("Enter " + n + " values.");
3.          for (int i = 0; i < n; i++) {
4.              System.out.print(": ");
5.              str = br.readLine();
6.              try {
7.                  t = Double.parseDouble(str);
8.              } catch (NumberFormatException exc) {
9.                  System.out.println("Invalid format");
10.                 t = 0.0;
11.             }
12.             sum += t;
13.         }
14.         avg = sum / n;
15.         System.out.println("Average is " + avg);
16.     }
17. }
```

Scanner

- Une autre façon de convertir une chaîne numérique dans son format binaire interne consiste à utiliser l'une des méthodes définies par `java.util.Scanner`
- `Scanner` lit l'entrée formaté et le convertit en sa forme binaire.
- `Scanner` peut être utilisé pour lire les entrées de diverses sources, y compris la console et les fichiers → utilisez `Scanner` pour lire une chaîne numérique entrée au clavier et attribuer sa valeur à une variable.
- Constructeur:

```
Scanner(InputStream from)
```

Exercices [1]

1. Écrivez un programme qui compte le nombre de lignes d'un fichier d'entrée.
2. Écrivez un programme qui compte le nombre de mots dans un fichier d'entrée.

Exercices [2]

- Écrivez un programme qui lit un fichier (fourni) contenant la liste des contacts. Chaque ligne comporte trois champs: prénom, nom, numéro de téléphone séparés par un tab (\t).
  - Définissez la classe Contact ayant les trois domaines mentionnés ci-dessus.
  - Lisez le fichier et placez-le dans une liste des contacts, puis ajoutez des opérations sur cette liste: Ajouter un nouveau contact, Modifier un contact, Supprimer un contact.
  - Enregistrer une liste des contacts dans un fichier de sortie.

Exercices [3]

- Écrivez un programme qui lit un fichier d'entrée (fourni) contenant les notes des étudiants du cours Java (chaque étudiant est stocké dans une ligne, tous les champs sont séparés par un tab (\t).
  - Définissez toutes les classes nécessaires.
  - Lire le fichier d'entrée et de mettre l'information dans une liste des étudiants.
  - Ajouter des opérations sur la liste des étudiants: gérer les étudiants, gérer les notes.
  - Enregistrez la liste dans un fichier de sortie avec la note moyenne (colonne TB).