BTS
BREVET DE TECHNICIEN SUPÉRIEUR
LYCÉE QUALIFIANT MOHAMED V
ESSAOUIRA

PROGRAMMATION ORIENTÉE OBJET (JAVA)

Les fichiers

1. Introduction

En java, les fichiers sont représentés par des objets particuliers appelés en anglais (Stream). Ce mot signifie flot en français.

Les streams sont représentés dans des classes définies dans le package java.io.

Il existe de nombreuses sortes de streams, qui peuvent être classés selon plusieurs critères:

→ Streams d'entrée (lecture) et streams de sortie (écriture)

→ Streams de caractères (texte) et streams de données binaires

→ Streams avec et sans tampon de données

→ …

2. Streams de données binaires

En java, les streams de données binaires dérivent des deux classes: → java.io.InputStream pour les entrées (lecture)

→ java.io.OutpuStream pour les sorties (écriture)

a. Ecriture dans un fichier de données binaires:

Les streams d'écriture pour les données binaires sont des sous classes de la classe java.io.OutputStream:

- DataOutputStream: Ecriture séquentielle dans un fichier binaire

-BufferedOutputStream: Ecriture des données à l'aide d'un tampon

- PrintStream: Ecriture de données avec conversion en octets en fonction du système hôte. Ce type de stream est dédié à l'affichage de valeurs sous forme de texte

- System.out est dérivée d'une classe PrintStream, voici quelques méthodes:

2. Streams de données binaires

Méthode	description
write() print() println() close()	Ecriture vers le stream Imprime les données sous forme d'un texte Imprime une ligne suivi d'un saut de ligne Ferme le stream

La classe DataOutpuStream admet le constructeur:

DataOutpuStream(FileOutputStream out): créer un stream d'écriture sur out

2. Streams de données binaires

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
class EcritureFichierBinaire
{
    public static void main(String[] arg) throws IOException
     {
        DataOutputStream ecrivain;
        ecrivain = new DataOutputStream (new FileOutputStream(arg[0]));
        ecrivain.writeUTF("Bonjour”);
        ecrivain.writeInt(5);
        ecrivain.writeChar(‘a’);
        ecrivain.writeBoolean(false);
        System.out.println(ecrivain.size()); // en octets
       ecrivain.close();
     }
}
```

1- Crée un fichier **EcritureFichierBinaire.java** avec le code.
2- Compile avec javac EcritureFichierBinaire.java.
3- Exécute avec java EcritureFichierBinaire fichier.bin.
4- Vérifie que le fichier binaire est bien créé.

2. Streams de données binaires

b. Lecture dans un fichier de données binaires:

Les streams d'entrée pour les données binaires sont des sous classes de la classe java.io.InputStream:

- DataInputStream: Lecture séquentielle dans un fichier binaire

-BufferedInputStream: Lecture des données à l'aide d'un tampon

- InputStream: Dispose d'un ensemble de méthodes

2. Streams de données binaires

Méthode	description
read() long skip(long n) int available() void mark(int p) void close()	Lit des données du stream Saute n octets du stream Renvoie le nombre d'octets disponible dans le stream Marque la position p dans le stream Ferme le stream

Read() lit un seul octet en entrée en renvoyant -1 lorsqu'il n'a y plus de données

La classe DataInpuStream admet le constructeur:

DataInpuStream(InputStream in): créer un stream d'entrée

2. Streams de données binaires

```
import java.io.*;
class LireFichierBinaire
{
   public static void main(String[] arg) throws
   IOException
      {
        DataInputStream lecture;
        lecture = new DataInputStream (new
        FileInputStream(arg[0]));
        System.out.println(lecture.readUTF());
        System.out.println(lecture.readInt());
        System.out.println(lecture.readChar());

   System.out.println(lecture.readBoolean());
        lecture.close();
      }
}
```

1- Crée un fichier **LireFichierBinaire.java** avec le code.
2- Compile avec javac LireFichierBinaire.java.
3- Exécute avec java LireFichierBinaire

3. Streams de caractères

Les streams caractères sont conçus pour la lecture et l'écriture de texte. Ils dérivent des deux classes abstraites: java.io.Reader pour les entrées et java.io.Writer pour les sorties.

a. Ecriture dans un fichier texte:

Pour écrire dans un fichier texte, il faut utiliser les streams d'écriture qui sont des sous-classes de la classe java.io.Writer, qui contient les sous classes suivantes:

- FileWriter: sous-classe particulière de OutputStreamWriter utilisant la taille de tampon par défaut.

- OutputStreamWriter: Convertir un stream de données binaires en stream de caractères.

- BufferedWriter: Ecriture de caractères à l'aide d'un tampon (à la fin de chaque ligne écrite, on utilise la méthode newLigne())

- PrintWriter: Ecrire des caractères formatés (la méthode println() permet d'écrire sur le fichier texte).

3. Streams de caractères

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;
class EcrireFichierTexte
{
   public static void main(String[] arg) throws IOException
     {
       PrintWriter ecrivain;
       int i=10;
       ecrivain = new PrintWriter (new FileOutputStream(arg[0]));
       ecrivain.println("Bonjour");
       ecrivain.println(" Comment cela va-t-il?");
       ecrivain.println(" On peut mettre des entiers");
       ecrivain.println(i);
       ecrivain.close();
     }
}
```

3. Streams de caractères

b. Lecture dans un fichier texte:

Pour lire dans un fichier texte, on doit utiliser des sous-classes de la classe java.io.Reader. Deux méthodes de cette classe sont abstraites:

- read(char[]cbuf,int off,int len): permet de lire **len** caractères et de les placer dans le tableau **cbuf**, à partir de l'indice **off**.

- close(): ferme le stream.

Toutes les classes dérivées de Reader redéfinissent donc obligatoirement ces deux méthodes.

- FileReader: sous-classe particulière de InputStreamReader utilisant une taille de tampon par défaut.

- BufferedReader: lecture de caractères à l'aide d'un tampon.

3. Streams de caractères

Premier exemple:

Il s'agit de lire un fichier texte ligne par ligne et de reproduire ce qui est lu directement à l'écran.

```
import java.io.*;
class LireLigne
{ public static void main(String[] arg) throws IOException
    {
      BufferedReader lectureAvecBuffer=null;
      String ligne;
      try{
            lectureAvecBuffer=new BufferedReader(new
      FileReader(''fichier.txt''));
      }
      catch(FileNotFoundException exc){
            System.out.println(" Erreur d'ouverture");
      }
       while((ligne=lectureAvecBuffer.readLine())!=null)
      System.out.println(ligne);
       lectureAvecBuffer.close();
    }
}
```

3. Streams de caractères

Deuxième exemple:

Il s'agit de lire des entiers dans un fichier ne contenant que des entiers et d'en faire la somme.

```
import java.io.*;
class LireEntiers
{
  public static void main(String[] arg) throws
  IOException
    {
      FileReader fichier=new
  FileReader(''fichier.txt'');
      int somme=0;
      StreamTokenizer entree=new
      StreamTokenizer(fichier);

  while(entree.nextToken()==StreamTokenizer.TT_N
  UMBER)
      somme+=(int)entree.nval;
       System.out.println('' La somme
  est:'' +somme);
       fichier.close();
    }
```

3. Streams de caractères

Troisième exemple:

Il s'agit de lire un fichier texte de nom « original.txt » et copie ce fichier dans un autre fichier de nom « copie.txt ». Nous allons utiliser les streams FileReader et FileWriter.

```
import java.io.*;
class CopieTxt
{
   public static void main(String[] arg) throws
   IOException
     {
       FileReader entree=new
   FileReader("original.txt");
       FileWriter sortie=new FileWriter("copie.txt");
       int c;

       while((c.entree.read())!=-1)
       sortie.write(c);
        entree.close();
        sortie.close();
     }
}
```

La classe Date

4. Exemple de la classe date

```java
import java.util.Date;          // Importation de la classe Date
import java.text.SimpleDateFormat; // Pour formater la date

public class ExempleDate {
    public static void main(String[] args) {
        // Création d'un objet Date qui représente la date et l'heure actuelles
        Date maintenant = new Date();

        // Affichage brut de la date
        System.out.println("Date brute : " + maintenant);

        // Formatage de la date dans un format lisible
        SimpleDateFormat formatDate = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        System.out.println("Date formatée : " + formatDate.format(maintenant));

        // Création d'une date spécifique (exemple : 1er janvier 2023 à 12h00)
        Date dateSpecifique = new Date(123, 0, 1, 12, 0, 0); // Année 123 = 2023 (car 1900 + 123)
        System.out.println("Date spécifique : " + formatDate.format(dateSpecifique));
    }
}
```

```
Date brute : Mon Mar 11 14:30:45 GMT 2024
Date formatée : 11/03/2024 14:30:45
Date spécifique : 01/01/2023 12:00:00
```

`java.util.Date`	Représente une date et heure
`java.util.Calendar`	Manipulation avancée des dates
`java.text.SimpleDateFormat`	Formatage et parsing des dates
`java.time.LocalDate`	Gère uniquement la **date** (sans heure)
`java.time.LocalTime`	Gère uniquement l'**heure** (sans date)
`java.time.LocalDateTime`	Gère la **date et l'heure**
`java.time.ZonedDateTime`	Gère les **fuseaux horaires**
`java.time.format.DateTimeFormatter`	Formatage des dates modernes (`java.time`)

Créer une date actuelle

```
import java.util.Date;
Date maintenant = new Date();
System.out.println(maintenant);
```

Créer une date spécifique

```
import java.util.Calendar;
Calendar calendrier = Calendar.getInstance();
calendrier.set(2023, Calendar.JANUARY, 1);
System.out.println(calendrier.getTime());
```

Formater une date

```
import java.util.Date;
import java.text.SimpleDateFormat;

SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
String dateFormatee = format.format(new Date());
System.out.println(dateFormatee);
```