

# Entrada y Salida

Manuel J. Molino Milla    Luis Molina Garzón

IES Virgen del Carmen

Departamento de Informática

19 de marzo de 2015

# Logo



Figura : Logo Java

# Contenido

Clase File

Stream

- bytes stream

- character stream

Random Access Files

I/O Formateado

File I/O in JDK 1.7

- Interface `java.nio.file.Path`

# Contenido

Clase File

Stream

bytes stream

character stream

Random Access Files

I/O Formateado

File I/O in JDK 1.7

Interface `java.nio.file.Path`

# Contenido

Clase File

Stream

bytes stream

character stream

Random Access Files

I/O Formateado

File I/O in JDK 1.7

Interface `java.nio.file.Path`

# Contenido

Clase File

Stream

bytes stream

character stream

Random Access Files

I/O Formateado

File I/O in JDK 1.7

Interface `java.nio.file.Path`

# Contenido

Clase File

Stream

- bytes stream

- character stream

Random Access Files

I/O Formateado

File I/O in JDK 1.7

- Interface `java.nio.file.Path`

# Introducción

JDK 1.0 Introduce el paquete *java.io*, I/O basada en stream

JDK 1.4 Introduce el paquete *java.nio*, I/O basada en buffer

JDK 1.5 Introduce I/O de texto formateado con nuevas clases  
*Scanner*, *Formatter* o *printf*

JDK 1.7 mediante *NIO.2* con I/O no bloqueante.



# Introducción

JDK 1.0 Introduce el paquete *java.io*, I/O basada en stream

JDK 1.4 Introduce el paquete *java.nio*, I/O basada en buffer

JDK 1.5 Introduce I/O de texto formateado con nuevas clases  
*Scanner*, *Formatter* o *printf*

JDK 1.7 mediante *NIO.2* con I/O no bloqueante.

# Introducción

JDK 1.0 Introduce el paquete *java.io*, I/O basada en stream

JDK 1.4 Introduce el paquete *java.nio*, I/O basada en buffer

JDK 1.5 Introduce I/O de texto formateado con nuevas clases  
*Scanner*, *Formatter* o *printf*

JDK 1.7 mediante *NIO.2* con I/O no bloqueante.

# Introducción

- JDK 1.0 Introduce el paquete *java.io*, I/O basada en stream
- JDK 1.4 Introduce el paquete *java.nio*, I/O basada en buffer
- JDK 1.5 Introduce I/O de texto formateado con nuevas clases *Scanner*, *Formatter* o *printf*
- JDK 1.7 mediante *NIO.2* con I/O no bloqueante.

# Introducción

- JDK 1.0 Introduce el paquete *java.io*, I/O basada en stream
- JDK 1.4 Introduce el paquete *java.nio*, I/O basada en buffer
- JDK 1.5 Introduce I/O de texto formateado con nuevas clases *Scanner*, *Formatter* o *printf*
- JDK 1.7 mediante *NIO.2* con I/O no bloqueante.

# Introducción


- JDK 1.0 Introduce el paquete *java.io*, I/O basada en stream
- JDK 1.4 Introduce el paquete *java.nio*, I/O basada en buffer
- JDK 1.5 Introduce I/O de texto formateado con nuevas clases *Scanner*, *Formatter* o *printf*
- JDK 1.7 mediante *NIO.2* con I/O no bloqueante.

# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '\' como separador de directorio; mientras que Unix/Mac usan '/'
- ▶ Windows usa '.' as como separador de la ruta de archivos, mientras que Unix/Mac usan '.'.
- ▶ Windows usa '\r\n' como fin de fichero; mientras que Unix usa '\n' y Mac usa '\r'.
- ▶ 'C:\' or '\ ' es el directorio raíz. Y en Unix/Mac es '/'
- ▶ El *path* se puede determinar de forma absoluta */home/usuario/fichero* o usar *./* o *..* como directorio actual o directorio padre, ejemplo: *./fichero*
- ▶ Constructores:

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```

```
public File(URI uri)
// Construyen un File con file-URI "file:///..."
```



# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '\' como separador de directorio; mientras que Unix/Mac usan '/'
- ▶ Windows usa '.' as como separador de la ruta de archivos, mientras que Unix/Mac usan '.'.
- ▶ Windows usa '\r\n' como fin de fichero; mientras que Unix usa '\n' y Mac usa '\r'.
- ▶ 'C:\' or '\ ' es el directorio raíz. Y en Unix/Mac es '/'
- ▶ El *path* se puede determinar de forma absoluta */home/usuario/fichero* o usar *./* o *..* como directorio actual o directorio padre, ejemplo: *./fichero*
- ▶ Constructores:

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```


```
public File(URI uri)
// Construyen un File con file-URI "file:///..."
```

# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '\' como separador de directorio; mientras que Unix/Mac usan '/'
- ▶ Windows usa ':' as como separador de la ruta de archivos, mientras que Unix/Mac usan '.'.
- ▶ Windows usa '\r\n' como fin de fichero; mientras que Unix usa '\n' y Mac usa '\r'.
- ▶ 'C:\' or '\ ' es el directorio raíz. Y en Unix/Mac es '/'
- ▶ El *path* se puede determinar de forma absoluta */home/usuario/fichero* o usar *./* o *..* como directorio actual o directorio padre, ejemplo: *./fichero*
- ▶ Constructores:

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```

```
public File(URI uri)
// Construyen un File con file-URI "file:///..."
```






# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '`\`' como separador de directorio; mientras que Unix/Mac usan '`/`'
- ▶ Windows usa '`:`' as como separador de la ruta de archivos, mientras que Unix/Mac usan '`:`'.
- ▶ Windows usa '`\r\n`' como fin de fichero; mientras que Unix usa '`\n`' y Mac usa '`\r`'.
- ▶ '`C:\`' or '`\`' es el directorio raíz. Y en Unix/Mac es '`/`'
- ▶ El *path* se puede determinar de forma absoluta `/home/usuario/fichero` o usar `./` o `..` como directorio actual o directorio padre, ejemplo: `./fichero`
- ▶ Constructores:

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```

```
public File(URI uri)
// Construyen un File con file-URI "file:///..."
```




# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '`\`' como separador de directorio; mientras que Unix/Mac usan '`/`'
- ▶ Windows usa '`:`' as como separador de la ruta de archivos, mientras que Unix/Mac usan '`:`'.
- ▶ Windows usa '`\r\n`' como fin de fichero; mientras que Unix usa '`\n`' y Mac usa '`\r`'.
- ▶ '`C:\`' or '`\`' es el directorio raíz. Y en Unix/Mac es '`/`'
- ▶ El *path* se puede determinar de forma absoluta `/home/usuario/fichero` o usar `.` o `..` como directorio actual o directorio padre, ejemplo: `./fichero`
- ▶ Constructores:

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```


```
public File(URI uri)
// Construyen un File con file-URI "file:///..."
```



# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '`\`' como separador de directorio; mientras que Unix/Mac usan '`/`'
- ▶ Windows usa '`:`' as como separador de la ruta de archivos, mientras que Unix/Mac usan '`:`'.
- ▶ Windows usa '`\r\n`' como fin de fichero; mientras que Unix usa '`\n`' y Mac usa '`\r`'.
- ▶ '`C:\`' or '`\`' es el directorio raíz. Y en Unix/Mac es '`/`'
- ▶ El *path* se puede determinar de forma absoluta `/home/usuario/fichero` o usar `.` o `..` como directorio actual o directorio padre, ejemplo: `./fichero`
- ▶ Constructores:


```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```

```
public File(URI uri)
// Construyen un File con file-URI "file:///."

```

# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '`\`' como separador de directorio; mientras que Unix/Mac usan '`/`'
- ▶ Windows usa '`:`' as como separador de la ruta de archivos, mientras que Unix/Mac usan '`:`'.
- ▶ Windows usa '`\r\n`' como fin de fichero; mientras que Unix usa '`\n`' y Mac usa '`\r`'.
- ▶ '`C:\`' or '`\`' es el directorio raíz. Y en Unix/Mac es '`/`'
- ▶ El *path* se puede determinar de forma absoluta `/home/usuario/fichero` o usar `.` o `..` como directorio actual o directorio padre, ejemplo: `./fichero`
- ▶ Constructores:


```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```

```
public File(URI uri)
// Construyen un File con file-URI "file://./." 
```

# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '`\`' como separador de directorio; mientras que Unix/Mac usan '`/`'
- ▶ Windows usa '`:`' as como separador de la ruta de archivos, mientras que Unix/Mac usan '`:`'.
- ▶ Windows usa '`\r\n`' como fin de fichero; mientras que Unix usa '`\n`' y Mac usa '`\r`'.
- ▶ '`C:\`' or '`\`' es el directorio raíz. Y en Unix/Mac es '`/`'
- ▶ El *path* se puede determinar de forma absoluta `/home/usuario/fichero` o usar `.` o `..` como directorio actual o directorio padre, ejemplo: `./fichero`
- ▶ Constructores:

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```


```
public File(URI uri)
// Construyen un File con file-URI "file:///."

```

# Class java.io.File (Pre-JDK 7)

- ▶ Representa tanto un fichero como un directorio.
- ▶ Windows usa '`\`' como separador de directorio; mientras que Unix/Mac usan '`/`'
- ▶ Windows usa '`:`' as como separador de la ruta de archivos, mientras que Unix/Mac usan '`:`'.
- ▶ Windows usa '`\r\n`' como fin de fichero; mientras que Unix usa '`\n`' y Mac usa '`\r`'.
- ▶ '`C:\`' or '`\`' es el directorio raíz. Y en Unix/Mac es '`/`'
- ▶ El *path* se puede determinar de forma absoluta `/home/usuario/fichero` o usar `.` o `..` como directorio actual o directorio padre, ejemplo: `./fichero`
- ▶ Constructores:

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Construyen un File dado el path
```

```
public File(URI uri)
// Construyen un File con file-URI "file:///..."
```



# Ejemplo

## Ejemplos básicos

```
File file = new File("in.txt"); //con ruta relativa
// con ruta absoluta
File file = new File("d:\\myproject\\java\\Hello.java");
File dir  = new File("c:\\temp");    // Un directorio
```

Para el caso de una aplicación que distribuimos como jar

```
java.net.URL url = this.getClass().getResource("icon.png");
```

# Ejemplo

## Ejemplos básicos

```
File file = new File("in.txt"); //con ruta relativa  
// con ruta absoluta  
File file = new File("d:\\myproject\\java\\Hello.java");  
File dir  = new File("c:\\temp");    // Un directorio
```

Para el caso de una aplicación que distribuimos como jar

```
java.net.URL url = this.getClass().getResource("icon.png");
```



# Verificando propiedades de un archivo o directorio

```
public boolean exists()           // Testea si archivo/directorio existe.
public long length()              // Devuelve la longitud del fichero.
public boolean isDirectory()      // Tes directorio.
public boolean isFile()           // Comprueba si es un archivo.
public boolean canRead()          // Comprueba si tiene permiso de lectura.
public boolean canWrite()         // Comprueba si tiene permiso de escritura.
public boolean delete()           // Borra el archivo/directorio.
public void deleteOnExit()        // Borra el archivo/directorio
//cuando el programa finalice..
public boolean renameTo(File dest) // Renombra el archivo.
public boolean mkdir()            // Crea el directorio.
public String[] list()            // Lista el contenido del directorio.
public File[] listFiles()         // Lista el contenido del directorio.
```

## Ejemplo listado recursivo de archivos

```
import java.io.File;
public class ListDirectoryRecursive {
    public static void main(String[] args) {
        File dir = new File("d:\\myproject\\test");
        listRecursive(dir);
    }

    public static void listRecursive(File dir) {
        if (dir.isDirectory()) {
            File[] items = dir.listFiles();
            for (File item : items) {
                System.out.println(item.getAbsolutePath());
                if (item.isDirectory()) listRecursive(item);
            }
        }
    }
}
```

# Listado de archivos con filtro

```
public String[] list(FilenameFilter filter)
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
//FileFilter usa el siguiente método:
public boolean accept(File dirName, String fileName)
```

## Ejemplo listado recursivo de archivos

```
// Listar ficheros terminados ".java"
import java.io.File;
import java.io FilenameFilter;
public class ListDirectoryWithFilter {
    public static void main(String[] args) {
        File dir = new File(".");    // current working directory
        if (dir.isDirectory()) {
            // List only files that meet the filtering criteria
            // programmed in accept() method of FilenameFilter.
            String[] files = dir.list(new FilenameFilter() {
                public boolean accept(File dir, String file) {
                    return file.endsWith(".java");
                }
            }); // clase interna de FilenameFilter
            for (String file : files) {
                System.out.println(file);
            }
        }
    }
}
```

# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa, . . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  - ▶ Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  - ▶ Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  - ▶ Cerrar del input/output stream.

# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  - ▶ Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  - ▶ Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  - ▶ Cerrar del input/output stream.

# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  - ▶ Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  - ▶ Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  - ▶ Cerrar del input/output stream.

# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  - Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  - Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  - Cerrar del input/output stream.



# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  - Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  - Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  - Cerrar del input/output stream.

# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  - Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  - Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  - Cerrar del input/output stream.

# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  1. Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  2. Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  3. Cerrar del input/output stream.

# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  1. Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  2. Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  3. Cerrar del input/output stream.

# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  1. Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  2. Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  3. Cerrar del input/output stream.

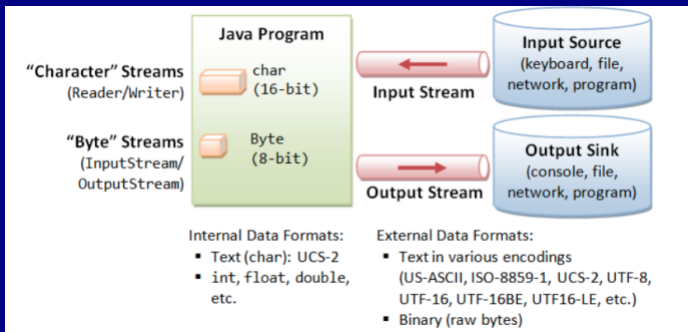
# Introducción a los stream

- ▶ Los programas leen datos del teclado, ficheros, de la red, de la memoria, . . . .
- ▶ Luego se envían al monitor, ficheros, a otro programa,. . . .
- ▶ Ej Java la I/O se maneja con *stream*
- ▶ Un *stream* es un flujo secuencial y contiguo de datos.
- ▶ Java recibe los datos mediante un *input stream*
- ▶ Y envía los datos mediante un *output stream*
- ▶ Las operaciones implican tres pasos:
  1. Abrir un input/output stream asociado con un dispositivo físico, construyendo el oportuno stream.
  2. Leer desde input stream abierto hasta encontrar el final del stream, o escribir en el output stream abierto (y opcionalmente volcado al buffer -flush-).
  3. Cerrar del input/output stream.

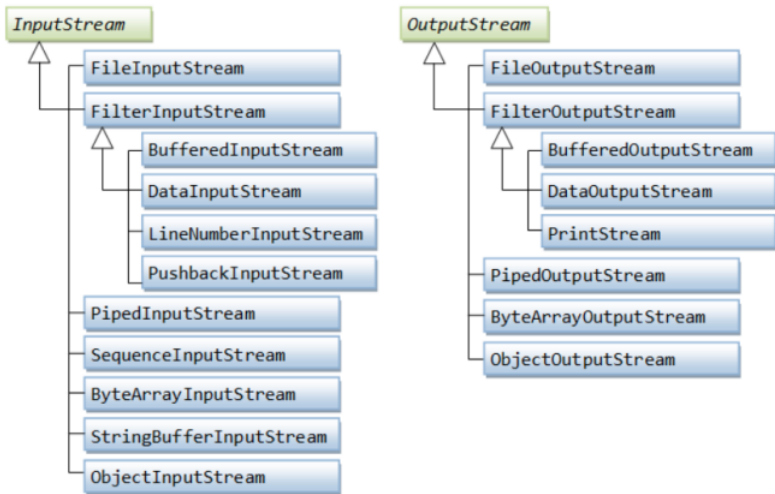
# Caracteres stream y byte stream

Java almacena de forma interna los caracteres usando 16-bit UCS-2. Pero la fuente de datos puede almacenar usando codificaciones diferentes como US-ASCII, ISO-8859-x, UTF-8, UTF-16, ...

Java necesita diferenciar entre I/O basada en **bytes**: procesamiento I/O raw bytes o binary data y en **caracteres** usando dos bytes.



# InputStream y OutputStream





# Leyendo de un InputStream

La clase *InputStream* es *abstracta* y declara un método *read*:

*public abstract int read() throws IOException*

-Devuelve:

- Los bytes leídos en un rango de 0 a 255.
- -1 si detecta el final del stream
- IOException si hay un error.

Otras variantes de *read*

```
public int read(byte[] bytes, int offset, int length)
    throws IOException
// Lee "length" numero de bytes, almacena desde
//el offset del índice.
public int read(byte[] bytes) throws IOException
// Lo mismo que read(bytes, 0, bytes.length)
```

# Leyendo de un InputStream

La clase *InputStream* es *abstracta* y declara un método *read*:

*public abstract int read() throws IOException*

-Devuelve:

- ▶ Los bytes leídos en un rango de 0 a 255.
- ▶ -1 si detecta el final del stream
- ▶ IOException si hay un error.

Otras variantes de *read*

```
public int read(byte[] bytes, int offset, int length)
    throws IOException
// Lee "length" numero de bytes, almacena desde
//el offset del índice.
public int read(byte[] bytes) throws IOException
// Lo mismo que read(bytes, 0, bytes.length)
```

# Leyendo de un InputStream

La clase *InputStream* es *abstracta* y declara un método *read*:

*public abstract int read() throws IOException*

-Devuelve:

- ▶ Los bytes leídos en un rango de 0 a 255.
- ▶ **-1** si detecta el final del stream
- ▶ *IOException* si hay un error.

Otras variantes de *read*

```
public int read(byte[] bytes, int offset, int length)
    throws IOException
// Lee "length" numero de bytes, almacena desde
//el offset del índice.
public int read(byte[] bytes) throws IOException
// Lo mismo que read(bytes, 0, bytes.length)
```

# Leyendo de un InputStream

La clase *InputStream* es *abstracta* y declara un método *read*:

*public abstract int read() throws IOException*

-Devuelve:

- ▶ Los bytes leídos en un rango de 0 a 255.
- ▶ **-1** si detecta el final del stream
- ▶ *IOException* si hay un error.

Otras variantes de *read*

```
public int read(byte[] bytes, int offset, int length)
    throws IOException
```

```
// Lee "length" numero de bytes, almacena desde
//el offset del índice.
```

```
public int read(byte[] bytes) throws IOException
// Lo mismo que read(bytes, 0, bytes.length)
```

# Leyendo de un InputStream

La clase *InputStream* es *abstracta* y declara un método *read*:

*public abstract int read() throws IOException*

-Devuelve:

- ▶ Los bytes leídos en un rango de 0 a 255.
- ▶ **-1** si detecta el final del stream
- ▶ *IOException* si hay un error.

Otras variantes de *read*

```
public int read(byte[] bytes, int offset, int length)
    throws IOException
```

```
// Lee "length" numero de bytes, almacena desde
//el offset del índice.
```

```
public int read(byte[] bytes) throws IOException
// Lo mismo que read(bytes, 0, bytes.length)
```

# Leyendo de un InputStream

La clase *InputStream* es *abstracta* y declara un método *read*:

*public abstract int read() throws IOException*

-Devuelve:

- ▶ Los bytes leídos en un rango de 0 a 255.
- ▶ **-1** si detecta el final del stream
- ▶ *IOException* si hay un error.

Otras variantes de *read*

```
public int read(byte[] bytes, int offset, int length)
    throws IOException
```

```
// Lee "length" numero de bytes, almacena desde
//el offset del índice.
```

```
public int read(byte[] bytes) throws IOException
// Lo mismo que read(bytes, 0, bytes.length)
```

# Escribiendo en un OutputStream

La clase *OutputStream* es *abstracta* y declara un método *write*:

```
public void abstract void write(int unsignedByte) throws  
IOException
```

Otras variantes de *write*

```
public void write(byte[] bytes, int offset, int length)  
    throws IOException  
// Escribe "length" numero de bytes desde el  
//offset del índice.  
public void write(byte[] bytes) throws IOException  
// Lo mismo que write(bytes, 0, bytes.length)
```

# Escribiendo en un OutputStream

La clase *OutputStream* es *abstracta* y declara un método *write*:

```
public void abstract void write(int unsignedByte) throws  
IOException
```

Otras variantes de *write*

```
public void write(byte[] bytes, int offset, int length)  
    throws IOException  
// Escribe "length" numero de bytes desde el  
//offset del índice.  
public void write(byte[] bytes) throws IOException  
// Lo mismo que write(bytes, 0, bytes.length)
```



# Apertura y cierre de stream

Es buena práctica cerrar el *stream* en una clausula *finally*

```
FileInputStream in = null;
.....
try {
    in = new FileInputStream(...);  // Open stream
    .....
    .....
} catch (IOException ex) {
    ex.printStackTrace();
} finally {  // always close the I/O streams
    try {
        if (in != null) in.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

JDK 1.7 introduce una nueva sintaxis *try-with-resources*,  
qué automáticamente cierra todos los recursos:

```
try (FileInputStream in = new FileInputStream(...)) {
```

# Apertura y cierre de stream

Es buena práctica cerrar el *stream* en una clausula *finally*

```
FileInputStream in = null;
.....
try {
    in = new FileInputStream(...); // Open stream
    .....
    .....
} catch (IOException ex) {
    ex.printStackTrace();
} finally { // always close the I/O streams
    try {
        if (in != null) in.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

JDK 1.7 introduce una nueva sintaxis *try-with-resources*,  
qué automáticamente cierra todos los recursos:

```
try (FileInputStream in = new FileInputStream(...)) {
```

# Flushing OutputStream

*public void flush() throws IOException*

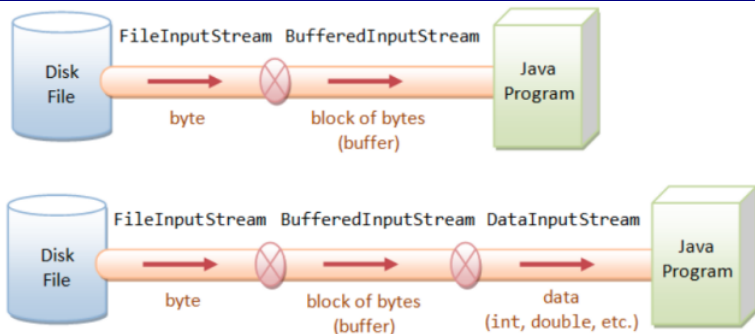
Fuerza al OutputStream a escribir los datos al dispositivo.

# Implementación de InputStream/OutputStream

*InputStream/OutputStream* es una clase abstracta, no se pueden crear objetos de la misma, en función del tipo de dispositivo usaremos la mas conveniente.

Ejemplo, para un archivo usaremos *FileInputStream* o *FileOutputStream*

# Encadenando stream



# BufferedInputStream & BufferedOutputStream

- ▶ Lee byte a byte en cada llamada.
- ▶ El método *read* de *InputStream* es muy ineficiente.
- ▶ Mejor es usar un bloque de byte en la lectura/escritura.
- ▶ Esto se consigue con un *buffer*
- ▶ Se realiza una operación de I/O desde el dispositivo al buffer de la memoria.

```
FileInputStream fileIn = new FileInputStream("in.dat");  
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);  
DataInputStream dataIn = new DataInputStream(bufferIn);  
// or  
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```

# BufferedInputStream & BufferedOutputStream

- ▶ Lee byte a byte en cada llamada.
- ▶ El método *read* de *InputStream* es muy ineficiente.
- ▶ Mejor es usar un bloque de byte en la lectura/escritura.
- ▶ Esto se consigue con un *buffer*
- ▶ Se realiza una operación de I/O desde el dispositivo al buffer de la memoria.

```
FileInputStream fileIn = new FileInputStream("in.dat");  
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);  
DataInputStream dataIn = new DataInputStream(bufferIn);  
// or  
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```

# BufferedInputStream & BufferedOutputStream

- ▶ Lee byte a byte en cada llamada.
- ▶ El método *read* de *InputStream* es muy ineficiente.
- ▶ Mejor es usar un bloque de byte en la lectura/escritura.
- ▶ Esto se consigue con un *buffer*
- ▶ Se realiza una operación de I/O desde el dispositivo al buffer de la memoria.

```
FileInputStream fileIn = new FileInputStream("in.dat");  
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);  
DataInputStream dataIn = new DataInputStream(bufferIn);  
// or  
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```



# BufferedInputStream & BufferedOutputStream

- ▶ Lee byte a byte en cada llamada.
- ▶ El método *read* de *InputStream* es muy ineficiente.
- ▶ Mejor es usar un bloque de byte en la lectura/escritura.
- ▶ Esto se consigue con un *buffer*
- ▶ Se realiza una operación de I/O desde el dispositivo al buffer de la memoria.

```
FileInputStream fileIn = new FileInputStream("in.dat");  
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);  
DataInputStream dataIn = new DataInputStream(bufferIn);  
// or  
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```

# BufferedInputStream & BufferedOutputStream

- ▶ Lee byte a byte en cada llamada.
- ▶ El método *read* de *InputStream* es muy ineficiente.
- ▶ Mejor es usar un bloque de byte en la lectura/escritura.
- ▶ Esto se consigue con un *buffer*
- ▶ Se realiza una operación de I/O desde el dispositivo al buffer de la memoria.

```
FileInputStream fileIn = new FileInputStream("in.dat");  
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);  
DataInputStream dataIn = new DataInputStream(bufferIn);  
// or  
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```

# BufferedInputStream & BufferedOutputStream

- ▶ Lee byte a byte en cada llamada.
- ▶ El método *read* de *InputStream* es muy ineficiente.
- ▶ Mejor es usar un bloque de byte en la lectura/escritura.
- ▶ Esto se consigue con un *buffer*
- ▶ Se realiza una operación de I/O desde el dispositivo al buffer de la memoria.

```
FileInputStream fileIn = new FileInputStream("in.dat");  
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);  
DataInputStream dataIn = new DataInputStream(bufferIn);  
// or  
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```

# BufferedInputStream & BufferedOutputStream

- ▶ Lee byte a byte en cada llamada.
- ▶ El método *read* de *InputStream* es muy ineficiente.
- ▶ Mejor es usar un bloque de byte en la lectura/escritura.
- ▶ Esto se consigue con un *buffer*
- ▶ Se realiza una operación de I/O desde el dispositivo al buffer de la memoria.

```
FileInputStream fileIn = new FileInputStream("in.dat");  
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);  
DataInputStream dataIn = new DataInputStream(bufferIn);  
// or  
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```

# Data-Streams formateados: DataInputStream

Los usamos cuando estamos leyendo datos primitivos o String.

```
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```

# Data-Streams formateados: DataInputStream

Los usamos cuando estamos leyendo datos primitivos o String.

```
DataInputStream in = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("in.dat")));
```

# Métodos de Data-Streams

```
// Para datos primitivos
public final int readInt() throws IOException;           // Lee 4 bytes
public final double readDouble() throws IOException;    // Lee 8 bytes
public final byte readByte() throws IOException;
public final char readChar() throws IOException;
public final short readShort() throws IOException;
public final long readLong() throws IOException;
public final boolean readBoolean() throws IOException;  // Lee 1 byte.
public final float readFloat() throws IOException;
public final int readUnsignedByte() throws IOException; // Lee 1 byte [0, 255]
public final int readUnsignedShort() throws IOException; // Lee 2 bytes
[0, 65535]
public final void readFully(byte[] b, int off, int len) throws IOException;
public final void readFully(byte[] b) throws IOException;

// Strings
public final String readLine() throws IOException;
    // Lee una línea (hasta nueva línea),
    // convierte cada byte a char - no soporta unicode.
public final String readUTF() throws IOException;
    // lee con String UTF-encoded con los primeros bytes indicando la longitud
    // en bytes del UTF

public final int skipBytes(int n) // Salta a número de bytes
```

# Data-Streams formateados: DataOutputStream

Los usamos cuando estamos escribiendo datos primitivos o String.

```
DataOutputStream out = new DataOutputStream(  
    new BufferedOutputStream(  
        new FileOutputStream("out.dat"))
```



# Data-Streams formateados: DataOutputStream

Los usamos cuando estamos escribiendo datos primitivos o String.

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

# Métodos de Data-Streams

```
public final void writeInt(int i) throws IOException;    // escribe 4 bytes
public final void writeFloat(float f) throws IOException;
public final void writeDouble(double d) throws IOException; //
public final void writeByte(int b) throws IOException;    //
public final void writeShort(int s) throws IOException;    //
public final void writeLong(long l) throws IOException;
public final void writeBoolean(boolean b) throws IOException;
public final void writeChar(int i) throws IOException;

// String
public final void writeBytes(String str) throws IOException;
public final void writeChars(String str) throws IOException;
    // Escribe String como UCS-2 16-bit char, Big-endian
public final void writeUTF(String str) throws IOException;
    // Escribe String como UTF, 2 bytes indican longitud de UTF bytes

public final void write(byte[] b, int off, int len) throws IOException
public final void write(byte[] b) throws IOException
public final void write(int b) throws IOException
```

# Character Streams

- ▶ Java usa el conjunto de caracteres 16-bit UCS-2.
- ▶ Pero externamente se pueden guardar con otra codificación: US-ASCII, ISO-8859-x, UTF-8, UTF-16, ....
- ▶ Independientemente, cuando trabajamos con I/O debemos diferenciar entre procesamiento de *bytes* (raw) o I/O basado en caracteres cuando se procesa texto.
- ▶ Para esto tenemos las clases abstractas *Reader* y *Writer* las cuales implementan los métodos:

```
public abstract int read() throws IOException
public int read(char[] chars,int offset,int length) throws IOException
public int read(char[] chars) throws IOException
```

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars,int offset,int length) throws IOException
public void write(char[] chars) throws IOException
```

# Character Streams

- ▶ Java usa el conjunto de caracteres 16-bit UCS-2.
- ▶ Pero externamente se pueden guardar con otra codificación: US-ASCII, ISO-8859-x, UTF-8, UTF-16, ....
- ▶ Independientemente, cuando trabajamos con I/O debemos diferenciar entre procesamiento de *bytes* (raw) o I/O basado en caracteres cuando se procesa texto.
- ▶ Para esto tenemos las clases abstractas *Reader* y *Writer* las cuales implementan los métodos:

```
public abstract int read() throws IOException
public int read(char[] chars,int offset,int length) throws IOException
public int read(char[] chars) throws IOException
```

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars,int offset,int length) throws IOException
public void write(char[] chars) throws IOException
```

# Character Streams

- ▶ Java usa el conjunto de caracteres 16-bit UCS-2.
- ▶ Pero externamente se pueden guardar con otra codificación: US-ASCII, ISO-8859-x, UTF-8, UTF-16, ....
- ▶ Independientemente, cuando trabajamos con I/O debemos diferenciar entre procesamiento de *bytes* (raw) o I/O basado en caracteres cuando se procesa texto.
- ▶ Para esto tenemos las clases abstractas *Reader* y *Writer* las cuales implementan los métodos:

```
public abstract int read() throws IOException
public int read(char[] chars,int offset,int length) throws IOException
public int read(char[] chars) throws IOException
```

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars,int offset,int length) throws IOException
public void write(char[] chars) throws IOException
```

# Character Streams

- ▶ Java usa el conjunto de caracteres 16-bit UCS-2.
- ▶ Pero externamente se pueden guardar con otra codificación: US-ASCII, ISO-8859-x, UTF-8, UTF-16, ....
- ▶ Independientemente, cuando trabajamos con I/O debemos diferenciar entre procesamiento de *bytes* (raw) o I/O basado en caracteres cuando se procesa texto.
- ▶ Para esto tenemos las clases abstractas *Reader* y *Writer* las cuales implementan los métodos:

```
public abstract int read() throws IOException
public int read(char[] chars,int offset,int length) throws IOException
public int read(char[] chars) throws IOException
```

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars,int offset,int length) throws IOException
public void write(char[] chars) throws IOException
```

# Character Streams

- ▶ Java usa el conjunto de caracteres 16-bit UCS-2.
- ▶ Pero externamente se pueden guardar con otra codificación: US-ASCII, ISO-8859-x, UTF-8, UTF-16, ....
- ▶ Independientemente, cuando trabajamos con I/O debemos diferenciar entre procesamiento de *bytes* (raw) o I/O basado en caracteres cuando se procesa texto.
- ▶ Para esto tenemos las clases abstractas *Reader* y *Writer* las cuales implementan los métodos:

```
public abstract int read() throws IOException
public int read(char[] chars,int offset,int length) throws IOException
public int read(char[] chars) throws IOException
```

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars,int offset,int length) throws IOException
public void write(char[] chars) throws IOException
```

# Character Streams

- ▶ Java usa el conjunto de caracteres 16-bit UCS-2.
- ▶ Pero externamente se pueden guardar con otra codificación: US-ASCII, ISO-8859-x, UTF-8, UTF-16, ....
- ▶ Independientemente, cuando trabajamos con I/O debemos diferenciar entre procesamiento de *bytes* (raw) o I/O basado en caracteres cuando se procesa texto.
- ▶ Para esto tenemos las clases abstractas *Reader* y *Writer* las cuales implementan los métodos:

```
public abstract int read() throws IOException
public int read(char[] chars,int offset,int length) throws IOException
public int read(char[] chars) throws IOException
```

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars,int offset,int length) throws IOException
public void write(char[] chars) throws IOException
```



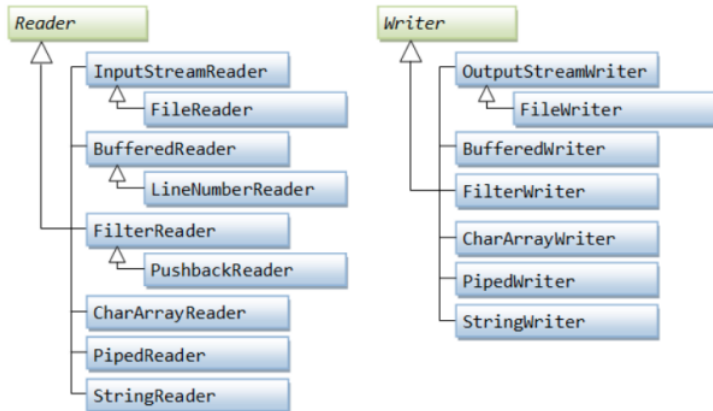
# Character Streams

- ▶ Java usa el conjunto de caracteres 16-bit UCS-2.
- ▶ Pero externamente se pueden guardar con otra codificación: US-ASCII, ISO-8859-x, UTF-8, UTF-16, ....
- ▶ Independientemente, cuando trabajamos con I/O debemos diferenciar entre procesamiento de *bytes* (raw) o I/O basado en caracteres cuando se procesa texto.
- ▶ Para esto tenemos las clases abstractas *Reader* y *Writer* las cuales implementan los métodos:

```
public abstract int read() throws IOException
public int read(char[] chars,int offset,int length) throws IOException
public int read(char[] chars) throws IOException
```

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars,int offset,int length) throws IOException
public void write(char[] chars) throws IOException
```

# Reader y Writer



# FileWriter & FileReader

Copiando un fichero de texto:

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Copia {
    public static void main(String[] args)
        throws IOException {
        File inputFile = new File("in.txt");
        File outputFile = new File("out.txt");
        FileReader in = new FileReader(inputFile);
        FileWriter out = new FileWriter(outputFile);
        int c;

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

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

# BufferedReader & BufferedWriter

- ▶ Se usan para envolver `FileWriter` y `FileReader`
- ▶ Mejora el rendimiento de ambos.
- ▶ Pues usamos un buffer de memoria.
- ▶ Además provee un nuevo método *`readLine()`*

# BufferedReader & BufferedWriter

- ▶ Se usan para envolver `FileWriter` y `FileReader`
- ▶ Mejora el rendimiento de ambos.
- ▶ Pues usamos un buffer de memoria.
- ▶ Además provee un nuevo método *`readLine()`*

# BufferedReader & BufferedWriter

- ▶ Se usan para envolver `FileWriter` y `FileReader`
- ▶ Mejora el rendimiento de ambos.
- ▶ Pues usamos un buffer de memoria.
- ▶ Además provee un nuevo método *`readLine()`*

# BufferedReader & BufferedWriter

- ▶ Se usan para envolver `FileWriter` y `FileReader`
- ▶ Mejora el rendimiento de ambos.
- ▶ Pues usamos un buffer de memoria.
- ▶ Además provee un nuevo método *readLine()*

# Ejemplo BufferedReader & BufferedWriter

```
import java.io.*;
// Write a text message to an output file, then read it back.
// FileReader/FileWriter uses the default charset for file encoding.
public class BufferedFileReaderWriterJDK7 {
    public static void main(String[] args) {
        String strFilename = "out.txt";
        String message = "Hello, world!\nHello, world again!\n";

        // Print the default charset
        System.out.println(java.nio.charset.Charset.defaultCharset());

        try (BufferedWriter out = new BufferedWriter(new FileWriter(strFilename))) {
            out.write(message);
            out.flush();
        } catch (IOException ex) {
            ex.printStackTrace();
        }

        try (BufferedReader in = new BufferedReader(new FileReader(strFilename))) {
            String inLine;
            while ((inLine = in.readLine()) != null) { // exclude newline
                System.out.println(inLine);
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```



# PrintStream & PrintWriter

- ▶ La clase *PrintStream* y *PrintWriter* se usa para escribir texto formateado bajo *OutputStream*
- ▶ Tenemos métodos como *print*, *printf* o *format*

```
import java.io.*;
public class Print{
    public static void main(String[] arg) throws Exception{
        PrintStream output = new PrintStream(
            new FileOutputStream(new File("hola.txt")));
        output.println(true);
        output.println((int) 123);
        output.println((float) 123.456);
        output.printf("%.2f %n", 12.3698);
        output.close();
    }
}
```

# PrintStream & PrintWriter

- ▶ La clase *PrintStream* y *PrintWriter* se usa para escribir texto formateado bajo *OutputStream*
- ▶ Tenemos métodos como *print*, *printf* o *format*

```
import java.io.*;
public class Print{
    public static void main(String[] arg) throws Exception{
        PrintStream output = new PrintStream(
            new FileOutputStream(new File("hola.txt")));
        output.println(true);
        output.println((int) 123);
        output.println((float) 123.456);
        output.printf("%.2f %n", 12.3698);
        output.close();
    }
}
```

# PrintStream & PrintWriter

- ▶ La clase *PrintStream* y *PrintWriter* se usa para escribir texto formateado bajo *OutputStream*
- ▶ Tenemos métodos como *print*, *printf* o *format*

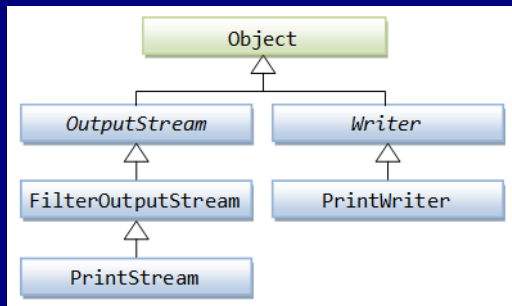
```
import java.io.*;
public class Print{
    public static void main(String[] arg) throws Exception{
        PrintStream output = new PrintStream(
            new FileOutputStream(new File("hola.txt")));
        output.println(true);
        output.println((int) 123);
        output.println((float) 123.456);
        output.printf("%.2f %n", 12.3698);
        output.close();
    }
}
```

# PrintStream & PrintWriter

- ▶ La clase *PrintStream* y *PrintWriter* se usa para escribir texto formateado bajo *OutputStream*
- ▶ Tenemos métodos como *print*, *printf* o *format*

```
import java.io.*;
public class Print{
    public static void main(String[] arg) throws Exception{
        PrintStream output = new PrintStream(
            new FileOutputStream(new File("hola.txt")));
        output.println(true);
        output.println((int) 123);
        output.println((float) 123.456);
        output.printf("%.2f %n", 12.3698);
        output.close();
    }
}
```

# PrintStream & PrintWriter



# Serialización y Object Streams

- ▶ *ObjectInputStream* y *ObjectOutputStream* nos permite leer y escribir objetos.
- ▶ Esos objetos pueden ser *ArrayList*, *Date* o cualquier objeto que creamos
- ▶ La *serialización* es el proceso consistente en convertir un objeto en un flujo de bytes (*stream*).
- ▶ La serialización de un objeto es necesaria bien cuando guardamos el estado del objeto en disco o lo enviámos a través de la red.
- ▶ Para que un objeto se pueda serializar debe implementar una de las dos siguientes interfaces: *java.io.Serializable* o *java.io.Externalizable*

```
public final Object readObject() throws IOException,  
    ClassNotFoundException;  
public final void writeObject(Object obj)  
    throws IOException;
```

# Serialización y Object Streams

- ▶ *ObjectInputStream* y *ObjectOutputStream* nos permite leer y escribir objetos.
- ▶ Esos objetos pueden ser *ArrayList*, *Date* o cualquier objeto que creamos
- ▶ La *serialización* es el proceso consistente en convertir un objeto en un flujo de bytes (*stream*).
- ▶ La serialización de un objeto es necesaria bien cuando guardamos el estado del objeto en disco o lo enviámos a través de la red.
- ▶ Para que un objeto se pueda serializar debe implementar una de las dos siguientes interfaces: *java.io.Serializable* o *java.io.Externalizable*

```
public final Object readObject() throws IOException,  
    ClassNotFoundException;  
public final void writeObject(Object obj)  
    throws IOException;
```

# Serialización y Object Streams

- ▶ *ObjectInputStream* y *ObjectOutputStream* nos permite leer y escribir objetos.
- ▶ Esos objetos pueden ser *ArrayList*, *Date* o cualquier objeto que creamos
- ▶ La *serialización* es el proceso consistente en convertir un objeto en un flujo de bytes (*stream*).
- ▶ La serialización de un objeto es necesaria bien cuando guardamos el estado del objeto en disco o lo enviámos a través de la red.
- ▶ Para que un objeto se pueda serializar debe implementar una de las dos siguientes interfaces: *java.io.Serializable* o *java.io.Externalizable*

```
public final Object readObject() throws IOException,  
    ClassNotFoundException;  
public final void writeObject(Object obj)  
    throws IOException;
```



# Serialización y Object Streams

- ▶ *ObjectInputStream* y *ObjectOutputStream* nos permite leer y escribir objetos.
- ▶ Esos objetos pueden ser *ArrayList*, *Date* o cualquier objeto que creamos
- ▶ La *serialización* es el proceso consistente en convertir un objeto en un flujo de bytes (*stream*).
- ▶ La serialización de un objeto es necesaria bien cuando guardamos el estado del objeto en disco o lo enviámos a través de la red.
- ▶ Para que un objeto se pueda serializar debe implementar una de las dos siguientes interfaces: *java.io.Serializable* o *java.io.Externalizable*

```
public final Object readObject() throws IOException,  
    ClassNotFoundException;  
public final void writeObject(Object obj)  
    throws IOException;
```

# Serialización y Object Streams

- ▶ *ObjectInputStream* y *ObjectOutputStream* nos permite leer y escribir objetos.
- ▶ Esos objetos pueden ser *ArrayList*, *Date* o cualquier objeto que creamos
- ▶ La *serialización* es el proceso consistente en convertir un objeto en un flujo de bytes (*stream*).
- ▶ La serialización de un objeto es necesaria bien cuando guardamos el estado del objeto en disco o lo enviámos a través de la red.
- ▶ Para que un objeto se pueda serializar debe implementar una de las dos siguientes interfaces: *java.io.Serializable* o *java.io.Externalizable*

```
public final Object readObject() throws IOException,  
    ClassNotFoundException;  
public final void writeObject(Object obj)  
    throws IOException;
```

# Serialización y Object Streams

- ▶ *ObjectInputStream* y *ObjectOutputStream* nos permite leer y escribir objetos.
- ▶ Esos objetos pueden ser *ArrayList*, *Date* o cualquier objeto que creamos
- ▶ La *serialización* es el proceso consistente en convertir un objeto en un flujo de bytes (*stream*).
- ▶ La serialización de un objeto es necesaria bien cuando guardamos el estado del objeto en disco o lo enviámos a través de la red.
- ▶ Para que un objeto se pueda serializar debe implementar una de las dos siguientes interfaces: *java.io.Serializable* o *java.io.Externalizable*

```
public final Object readObject() throws IOException,  
    ClassNotFoundException;  
public final void writeObject(Object obj)  
    throws IOException;
```

# Serialización y Object Streams

- ▶ *ObjectInputStream* y *ObjectOutputStream* nos permite leer y escribir objetos.
- ▶ Esos objetos pueden ser *ArrayList*, *Date* o cualquier objeto que creamos
- ▶ La *serialización* es el proceso consistente en convertir un objeto en un flujo de bytes (*stream*).
- ▶ La serialización de un objeto es necesaria bien cuando guardamos el estado del objeto en disco o lo enviámos a través de la red.
- ▶ Para que un objeto se pueda serializar debe implementar una de las dos siguientes interfaces: *java.io.Serializable* o *java.io.Externalizable*

```
public final Object readObject() throws IOException,  
    ClassNotFoundException;  
public final void writeObject(Object obj)  
    throws IOException;
```

# Ejemplos de ObjectOutputStream & ObjectInputStream

```
ObjectOutputStream out =  
    new ObjectOutputStream(  
        new BufferedOutputStream(  
            new FileOutputStream("object.ser")));  
out.writeObject("The current Date and Time is ");  
out.writeObject(new Date());  
out.flush();  
out.close();
```

```
ObjectInputStream in =  
    new ObjectInputStream(  
        new BufferedInputStream(  
            new FileInputStream("object.ser")));  
String str = (String)in.readObject();  
Date d = (Date)in.readObject(new Date());  
in.close();
```

# Ejemplos de ObjectOutputStream & ObjectInputStream

```
ObjectOutputStream out =  
    new ObjectOutputStream(  
        new BufferedOutputStream(  
            new FileOutputStream("object.ser")));  
out.writeObject("The current Date and Time is ");  
out.writeObject(new Date());  
out.flush();  
out.close();
```

```
ObjectInputStream in =  
    new ObjectInputStream(  
        new BufferedInputStream(  
            new FileInputStream("object.ser")));  
String str = (String)in.readObject();  
Date d = (Date)in.readObject(new Date());  
in.close();
```

# Serialización

- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long" (Advanced)*
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:
  - Ignorar el mensaje.
  - Añadir un id: *private static final long serialVersionUID = 1L;*
  - Usar la notación *@SuppressWarnings:*  
*@SuppressWarnings("serial") public class MyFrame extends JFrame { ..... }*

# Serialización

- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"* (Advanced)
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:
  - Ignorar el mensaje.
  - Añadir un id: *private static final long serialVersionUID = 1L;*
  - Usar la notación *@SuppressWarnings:*  
*@SuppressWarnings("serial") public class MyFrame extends JFrame { ..... }*



# Serialización

- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long" (Advanced)*
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:
  - Ignorar el mensaje.
  - Añadir un id: *private static final long serialVersionUID = 1L;*
  - Usar la notación *@SuppressWarnings:*  
*@SuppressWarnings("serial") public class MyFrame extends JFrame { ..... }*

# Serialización

- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"* (Advanced)
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:
  - Ignorar el mensaje.
  - Añadir un id: *private static final long serialVersionUID = 1L;*
  - Usar la notación *@SuppressWarnings:*  
*@SuppressWarnings("serial") public class MyFrame extends JFrame { ..... }*

# Serialización

- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"* (Advanced)
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:

Ignorar el mensaje.

Añadir un id: *private static final long serialVersionUID = 1L;*

Usar la notación *@SuppressWarnings:*

```
@SuppressWarnings("serial") public class MyFrame extends  
JFrame { ..... }
```

# Serialización

- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"* (Advanced)
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:
  1. Ignorar el mensaje.
  2. Añadir un id: *private static final long serialVersionUID = 1L;*
  3. Usar la notación *@SuppressWarnings*:  
*@SuppressWarnings("serial") public class MyFrame extends JFrame { ..... }*

# Serialización

- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"* (Advanced)
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:
  1. Ignorar el mensaje.
  2. Añadir un id: *private static final long serialVersionUID = 1L;*
  3. Usar la notación *@SuppressWarnings*:  
*@SuppressWarnings("serial") public class MyFrame extends JFrame { ..... }*

# Serialización

- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"* (Advanced)
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:
  1. Ignorar el mensaje.
  2. Añadir un id: *private static final long serialVersionUID = 1L;*
  3. Usar la notación *@SuppressWarnings*:  
*@SuppressWarnings("serial") public class MyFrame extends JFrame { ..... }*

# Serialización

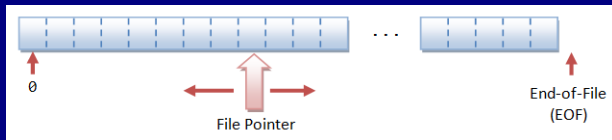
- ▶ Los datos primitivos y array, por defecto son serializables.
- ▶ Los campos estáticos no son serializables.
- ▶ Si queremos que ciertos campos no sean serializables usamos el modificador *transient*
- ▶ A veces aparece el mensaje *Warning Message "The serialization class does not declare a static final serialVersionUID field of type long"* (Advanced)
- ▶ Debido que algunas clases ya implementan la interfaz *Serializable*
- ▶ Para evitar este mensaje podemos hacer:
  1. Ignorar el mensaje.
  2. Añadir un id: *private static final long serialVersionUID = 1L;*
  3. Usar la notación *@SuppressWarnings:*  
*@SuppressWarnings("serial") public class MyFrame extends JFrame { ..... }*





## Acceso no secuencial de ficheros

- ▶ Los *stream* que hemos visto o son de escritura o de lectura.
- ▶ También existen *stream* de acceso secuencial.
- ▶ Valen tanto para la lectura como para la escritura.
- ▶ Lo que nos permiten modificar así como insertar nuevos datos.
- ▶ La clase a usar es la clase *RandomAccessFile*
- ▶ *RandomAccessFile* es como un gran array de bytes. Con un puntero localizado en la posición 0 al abrir el *stream*
- ▶ Dicho puntero avanza con la lectura de un número de bytes.



















# RandomAccessFile

## Constructores

```
RandomAccessFile f1 = new RandomAccessFile("filename", "r");  
RandomAccessFile f2 = new RandomAccessFile("filename", "rw");
```

## Métodos sobre el puntero

```
public void seek(long pos) throws IOException;  
// posiciona el puntero en nueva posición.  
public int skipBytes(int numBytes) throws IOException;  
// Desplaza el puntero una serie de bytes.  
public long getFilePointer() throws IOException;  
// Obtiene la posición del puntero  
public long length() throws IOException;  
// Devuelve el tamaño del fichero.
```

## Métodos de lectura/escritura

```
public int readInt() throws IOException;  
public double readDouble() throws IOException;  
public void writeInt(int i) throws IOException;  
public void writeDouble(double d) throws IOException;
```

# RandomAccessFile

## Constructores

```
RandomAccessFile f1 = new RandomAccessFile("filename", "r");  
RandomAccessFile f2 = new RandomAccessFile("filename", "rw");
```

## Métodos sobre el puntero

```
public void seek(long pos) throws IOException;  
// posiciona el puntero en nueva posición.  
public int skipBytes(int numBytes) throws IOException;  
// Desplaza el puntero una serie de bytes.  
public long getFilePointer() throws IOException;  
// Obtiene la posición del puntero  
public long length() throws IOException;  
// Devuelve el tamaño del fichero.
```

## Métodos de lectura/escritura

```
public int readInt() throws IOException;  
public double readDouble() throws IOException;  
public void writeInt(int i) throws IOException;  
public void writeDouble(double d) throws IOException;
```

# RandomAccessFile

## Constructores

```
RandomAccessFile f1 = new RandomAccessFile("filename", "r");  
RandomAccessFile f2 = new RandomAccessFile("filename", "rw");
```

## Métodos sobre el puntero

```
public void seek(long pos) throws IOException;  
// posiciona el puntero en nueva posición.  
public int skipBytes(int numBytes) throws IOException;  
// Desplaza el puntero una serie de bytes.  
public long getFilePointer() throws IOException;  
// Obtiene la posición del puntero  
public long length() throws IOException;  
// Devuelve el tamaño del fichero.
```

## Métodos de lectura/escritura

```
public int readInt() throws IOException;  
public double readDouble() throws IOException;  
public void writeInt(int i) throws IOException;  
public void writeDouble(double d) throws IOException;
```

# Ejemplo

```
import java.io.*;
public class TestRandomAccessFile{
    public static void main (String[]args) throws IOException {
// Create a random-access file
        RandomAccessFile inout = new RandomAccessFile ("inout.dat", "rw");
// Clear the file to destroy the old contents, if any
        inout.setLength (0);
// Write new integers to the file
        for (int i = 0; i < 200; i++)
            inout.writeInt (i);
// Display the current length of the file
        System.out.println ("Current file length is " + inout.length ());
// Retrieve the first number
        inout.seek (0);          // Move the file pointer to the beginning
        System.out.println ("The first number is " + inout.readInt ());
// Retrieve the second number
        inout.seek (1 * 4);      // Move the file pointer to the second number
        System.out.println ("The second number is " + inout.readInt ());
// Retrieve the tenth number
        inout.seek (9 * 4);      // Move the file pointer to the tenth number
        System.out.println ("The tenth number is " + inout.readInt ());
// Modify the eleventh number
        inout.writeInt (555);
// Append a new number
        inout.seek (inout.length ());    // Move the file pointer to the end
        inout.writeInt (999);
// Display the new length
        System.out.println ("The new length is " + inout.length ());
// Retrieve the new eleventh number
        inout.seek (10 * 4);      // Move the file pointer to the next number
        System.out.println ("The eleventh number is " + inout.readInt ());
        inout.close ();
    }
}
```

# Clase Scanner

- ▶ JDK 1.5 introduce `java.util.Scanner`.
- ▶ Parsea tokens usando diferentes métodos `nextInt()`, `nextByte()`, `nextShort()`, `nextLong()`, `nextFloat()`, `nextDouble()`, `nextBoolean()`, `next()` for `String`, y `nextLine()`
- ▶ Existen métodos `hasNextXxx()` para chequear la disponibilidad de la entrada.

# Clase Scanner

- ▶ JDK 1.5 introduce `java.util.Scanner`.
- ▶ Parsea tokens usando diferentes métodos *`nextInt()`*, *`nextByte()`*, *`nextShort()`*, *`nextLong()`*, *`nextFloat()`*, *`nextDouble()`*, *`nextBoolean()`*, *`next()` for `String`*, y *`nextLine()`*
- ▶ Existen métodos *`hasNextXxx()`* para chequear la disponibilidad de la entrada.

# Clase Scanner

- ▶ JDK 1.5 introduce `java.util.Scanner`.
- ▶ Parsea tokens usando diferentes métodos `nextInt()`, `nextByte()`, `nextShort()`, `nextLong()`, `nextFloat()`, `nextDouble()`, `nextBoolean()`, `next()` for `String`, y `nextLine()`
- ▶ Existen métodos `hasNextXxx()` para chequear la disponibilidad de la entrada.

# Clase Scanner

## Constructores

```
public Scanner(File source) throws FileNotFoundException
public Scanner(File source, String charsetName) throws FileNotFoundException
// Para System.in
public Scanner(InputStream source)
public Scanner(InputStream source, String charsetName)
// para un String
public Scanner(String source)
```

## Ejemplo

```
// Construye un Scanner para parsear un int desde teclado
Scanner in1 = new Scanner(System.in);
int i = in1.nextInt();

// Construye un Scanner para parsear los dobles de un fichero
Scanner in2 = new Scanner(new File("in.txt"));   FileNotFoundException
while (in2.hasNextDouble()) {
    double d = in2.nextDouble();
}

// Construye un Scanner para parsear string
Scanner in3 = new Scanner("This is the input text String");
while (in3.hasNext()) {
    String s = in3.next();
}
```



# Clase Scanner

## Constructores

```
public Scanner(File source) throws FileNotFoundException
public Scanner(File source, String charsetName) throws FileNotFoundException
// Para System.in
public Scanner(InputStream source)
public Scanner(InputStream source, String charsetName)
// para un String
public Scanner(String source)
```

## Ejemplo

```
// Construye un Scanner para parsear un int desde teclado
Scanner in1 = new Scanner(System.in);
int i = in1.nextInt();

// Construye un Scanner para parsear los dobles de un fichero
Scanner in2 = new Scanner(new File("in.txt"));  FileNotFoundException
while (in2.hasNextDouble()) {
    double d = in2.nextDouble();
}

// Construye un Scanner para parsear string
Scanner in3 = new Scanner("This is the input text String");
while (in3.hasNext()) {
    String s = in3.next();
}
```

# Scanner y el método useDelimiter()

- ▶ *useDelimiter (pattern)*
- ▶ Establece el delimitador para crear *tokens*
- ▶ Ejemplo:

```
import java.util.Scanner;

public class ScannerTokenizingText {
    public static void main(String[] args) {
        String text = "4231, Java Programming, 1000.00";
        Scanner scanner = new Scanner(text).useDelimiter("\\s*,\\s*");
        int checkNumber = scanner.nextInt();
        String description = scanner.next();
        float amount = scanner.nextFloat();
        System.out.printf("/***** Tokenizing Text *****/\\n\\n");
        System.out.printf("String to tokenize: %s\\n", text);
        System.out.printf("checkNumber: %d\\n", checkNumber);
        System.out.printf("description: %s\\n", description);
        System.out.printf("amount: %f", amount);
    }
}
```

# Scanner y el método useDelimiter()

- ▶ *useDelimiter (pattern)*
- ▶ Establece el delimitador para crear *tokens*
- ▶ Ejemplo:

```
import java.util.Scanner;

public class ScannerTokenizingText {
    public static void main(String[] args) {
        String text = "4231, Java Programming, 1000.00";
        Scanner scanner = new Scanner(text).useDelimiter("\\s*,\\s*");
        int checkNumber = scanner.nextInt();
        String description = scanner.next();
        float amount = scanner.nextFloat();
        System.out.printf("/***** Tokenizing Text *****/\\n\\n");
        System.out.printf("String to tokenize: %s\\n", text);
        System.out.printf("checkNumber: %d\\n", checkNumber);
        System.out.printf("description: %s\\n", description);
        System.out.printf("amount: %f", amount);
    }
}
```

# Scanner y el método useDelimiter()

- ▶ *useDelimiter (pattern)*
- ▶ Establece el delimitador para crear *tokens*
- ▶ Ejemplo:

```
import java.util.Scanner;

public class ScannerTokenizingText {
    public static void main(String[] args) {
        String text = "4231, Java Programming, 1000.00";
        Scanner scanner = new Scanner(text).useDelimiter("\\s*,\\s*");
        int checkNumber = scanner.nextInt();
        String description = scanner.next();
        float amount = scanner.nextFloat();
        System.out.printf("/***** Tokenizing Text *****/\\n\\n");
        System.out.printf("String to tokenize: %s\\n", text);
        System.out.printf("checkNumber: %d\\n", checkNumber);
        System.out.printf("description: %s\\n", description);
        System.out.printf("amount: %f", amount);
    }
}
```

# Scanner y el método useDelimiter()

- ▶ *useDelimiter (pattern)*
- ▶ Establece el delimitador para crear *tokens*
- ▶ Ejemplo:

```
import java.util.Scanner;

public class ScannerTokenizingText {
    public static void main(String[] args) {
        String text = "4231, Java Programming, 1000.00";
        Scanner scanner = new Scanner(text).useDelimiter("\\s*,\\s*");
        int checkNumber = scanner.nextInt();
        String description = scanner.next();
        float amount = scanner.nextFloat();
        System.out.printf("/***** Tokenizing Text *****/\\n\\n");
        System.out.printf("String to tokenize: %s\\n", text);
        System.out.printf("checkNumber: %d\\n", checkNumber);
        System.out.printf("description: %s\\n", description);
        System.out.printf("amount: %f", amount);
    }
}
```

# Scanner y el método useDelimiter()

- ▶ *useDelimiter (pattern)*
- ▶ Establece el delimitador para crear *tokens*
- ▶ Ejemplo:

```
import java.util.Scanner;

public class ScannerTokenizingText {
    public static void main(String[] args) {
        String text = "4231, Java Programming, 1000.00";
        Scanner scanner = new Scanner(text).useDelimiter("\\s*,\\s*");
        int checkNumber = scanner.nextInt();
        String description = scanner.next();
        float amount = scanner.nextFloat();
        System.out.printf("/***** Tokenizing Text *****/\\n\\n");
        System.out.printf("String to tokenize: %s\\n", text);
        System.out.printf("checkNumber: %d\\n", checkNumber);
        System.out.printf("description: %s\\n", description);
        System.out.printf("amount: %f", amount);
    }
}
```

# String.format

- ▶ Su compartamiento es similar al de *printf*
- ▶ Se usa un *patrón de formateo*
- ▶ Los parámetros separados por comas.
- ▶ Ejemplos:

```
int edad = 28;  
String nombre = "David";  
String patron = "El nombre de la persona es %s y tiene %d años";  
String resultado = String.format(patron,nombre,edad);  
System.out.print(resultado)  
//El nombre de la persona es David y tiene 28 años
```

```
int hora = 13;  
int minutos = 45;  
String nombre = "David";  
String patron = "%s ha accedido a las %d:%d h";  
String resultado = String.format(patron,nombre,hora,minutos);  
System.out.print(resultado);  
// David ha accedido a las 13:45 h
```

# String.format

- ▶ Su compartamiento es similar al de *printf*
- ▶ Se usa un *patrón de formateo*
- ▶ Los parámetros separados por comas.
- ▶ Ejemplos:

```
int edad = 28;
String nombre = "David";
String patron = "El nombre de la persona es %s y tiene %d años";
String resultado = String.format(patron,nombre,edad);
System.out.print(resultado)
//El nombre de la persona es David y tiene 28 años
```

```
int hora = 13;
int minutos = 45;
String nombre = "David";
String patron = "%s ha accedido a las %d:%d h";
String resultado = String.format(patron,nombre,hora,minutos);
System.out.print(resultado);
// David ha accedido a las 13:45 h
```



# String.format

- ▶ Su compartamiento es similar al de *printf*
- ▶ Se usa un *patrón de formateo*
- ▶ Los parámetros separados por comas.
- ▶ Ejemplos:

```
int edad = 28;
String nombre = "David";
String patron = "El nombre de la persona es %s y tiene %d años";
String resultado = String.format(patron,nombre,edad);
System.out.print(resultado)
//El nombre de la persona es David y tiene 28 años
```

```
int hora = 13;
int minutos = 45;
String nombre = "David";
String patron = "%s ha accedido a las %d:%d h";
String resultado = String.format(patron,nombre,hora,minutos);
System.out.print(resultado);
// David ha accedido a las 13:45 h
```

# String.format

- ▶ Su compartamiento es similar al de *printf*
- ▶ Se usa un *patrón de formateo*
- ▶ Los parámetros separados por comas.
- ▶ Ejemplos:

```
int edad = 28;
String nombre = "David";
String patron = "El nombre de la persona es %s y tiene %d años";
String resultado = String.format(patron,nombre,edad);
System.out.print(resultado)
//El nombre de la persona es David y tiene 28 años
```

```
int hora = 13;
int minutos = 45;
String nombre = "David";
String patron = "%s ha accedido a las %d:%d h";
String resultado = String.format(patron,nombre,hora,minutos);
System.out.print(resultado);
// David ha accedido a las 13:45 h
```

# String.format

- ▶ Su compartamiento es similar al de *printf*
- ▶ Se usa un *patrón de formateo*
- ▶ Los parámetros separados por comas.
- ▶ Ejemplos:

```
int edad = 28;
String nombre = "David";
String patron = "El nombre de la persona es %s y tiene %d años";
String resultado = String.format(patron,nombre,edad);
System.out.print(resultado)
//El nombre de la persona es David y tiene 28 años
```

```
int hora = 13;
int minutos = 45;
String nombre = "David";
String patron = "%s ha accedido a las %d:%d h";
String resultado = String.format(patron,nombre,hora,minutos);
System.out.print(resultado);
// David ha accedido a las 13:45 h
```

# String.format

- ▶ Su compartamiento es similar al de *printf*
- ▶ Se usa un *patrón de formateo*
- ▶ Los parámetros separados por comas.
- ▶ Ejemplos:

```
int edad = 28;
String nombre = "David";
String patron = "El nombre de la persona es %s y tiene %d años";
String resultado = String.format(patron,nombre,edad);
System.out.print(resultado)
//El nombre de la persona es David y tiene 28 años
```

```
int hora = 13;
int minutos = 45;
String nombre = "David";
String patron = "%s ha accedido a las %d:%d h";
String resultado = String.format(patron,nombre,hora,minutos);
System.out.print(resultado);
// David ha accedido a las 13:45 h
```

# String.format

- ▶ Su compartamiento es similar al de *printf*
- ▶ Se usa un *patrón de formateo*
- ▶ Los parámetros separados por comas.
- ▶ Ejemplos:

```
int edad = 28;
String nombre = "David";
String patron = "El nombre de la persona es %s y tiene %d años";
String resultado = String.format(patron,nombre,edad);
System.out.print(resultado)
//El nombre de la persona es David y tiene 28 años
```

```
int hora = 13;
int minutos = 45;
String nombre = "David";
String patron = "%s ha accedido a las %d:%d h";
String resultado = String.format(patron,nombre,hora,minutos);
System.out.print(resultado);
// David ha accedido a las 13:45 h
```

# Helper class java.nio.file.Paths

```
public static Path get(String first, String... more)
// Este método acepta varios argumentos).
// Los une formando un objeto Path.
// La localización del Path puede o no existir.
```

```
public static Path get(URI uri)
// Convierte el URI a un objeto Path.
```

Ejemplos:

```
Path p1 = Paths.get("in.txt");
Path p2 = Paths.get("c:\\myproejct\\java\\Hello.java");
Path p3 = Paths.get("/use/local");
```

# Ejemplo

```
import java.nio.file.*;
public class PathInfo {
    public static void main(String[] args) {
        // Windows
        Path path = Paths.get("D:\\myproject\\java\\test\\Hello.java");
        // Unix/Mac
        //Path path = Paths.get("/myproject/java/test/Hello.java");

        // Print Path Info
        System.out.println("toString:      " + path.toString());    // D:\myproject\java\test\Hello.java
        System.out.println("getFileName: " + path.getFileName());  // Hello.java
        System.out.println("getParent:   " + path.getParent());     // D:\myproject\java\test
        System.out.println("getRoot:     " + path.getRoot());       // D:\

        // root, level-0, level-1, ...
        int nameCount = path.getNameCount();
        System.out.println("getNameCount: " + nameCount);    // 4
        for (int i = 0; i < nameCount; ++i) {
            System.out.println("getName(" + i + "): " + path.getName(i)); // (0)myproject, (1)java,
        }                                                         // (2) test, (3) Hello.java
        System.out.println("subpath(0,2): " + path.subpath(0,2));  // myproject\java
        System.out.println("subpath(1,4): " + path.subpath(1,4));  // java\test\Hello.java
    }
}
```

# Helper Class java.nio.file.Files

```
public static long size(Path path) // Returns the size of the file

public static boolean exists(Path path, LinkOption... options)
// Verificas si el Path existse como un file/directory/symlink.
// si devuelve false si el file no existe
// LinkOption especifica como symlink deberían se meneados,
// ejemplo: NOFOLLOW_LINKS: no seguir enlaces.
public static boolean notExists(Path path, LinkOption... options) // ¿Existe?

public static boolean isDirectory(Path path, LinkOption... options) // ¿Directorio?
public static boolean isRegularFile(Path path, LinkOption... options) // ¿Fichero?
public static boolean isSymbolicLink(Path path) // ¿Enlace?

public static boolean isReadable(Path path) // ¿permiso lectura?
public static boolean isWritable(Path path) // ¿permiso escritura?
public static boolean isExecutable(Path path) // ¿permiso ejecución?
```



## Otros métodos

```
public static void delete(Path path) throws IOException
public static boolean deleteIfExists(Path path)
    throws IOException
```

```
public static Path copy(Path source, Path target,
    CopyOption... options) throws IOException
public static Path move(Path source, Path target,
    CopyOption... options) throws IOException
```

CopyOption: REPLACE\_EXISTING, COPY\_ATTRIBUTES,  
NOFOLLOW\_LINKS

## Otros métodos

```
public static void delete(Path path) throws IOException  
public static boolean deleteIfExists(Path path)  
    throws IOException
```

```
public static Path copy(Path source, Path target,  
    CopyOption... options) throws IOException  
public static Path move(Path source, Path target,  
    CopyOption... options) throws IOException
```

CopyOption: REPLACE\_EXISTING, COPY\_ATTRIBUTES,  
NOFOLLOW\_LINKS

## Otros métodos

```
public static void delete(Path path) throws IOException  
public static boolean deleteIfExists(Path path)  
    throws IOException
```

```
public static Path copy(Path source, Path target,  
    CopyOption... options) throws IOException  
public static Path move(Path source, Path target,  
    CopyOption... options) throws IOException
```

CopyOption: REPLACE\_EXISTING, COPY\_ATTRIBUTES,  
NOFOLLOW\_LINKS

# Otros métodos

```
public static byte[] readAllBytes(Path path) throws IOException
// Lee todos los bytes y devuelve byte[].
public static Path write(Path path, byte[] bytes,
    OpenOption... options) throws IOException
// Opciones spn: CREATE, TRUNCATE_EXISTING, y WRITE

public static List<String> readAllLines(Path path, Charset cs)
    throws IOException
// Lee todas las líneas.
// Terminador de línea puede ser "\n", "\r\n" or "\r".
public static Path write(Path path, Iterable<? extends CharSequence> lines,
    Charset cs, OpenOption... options) throws IOException
```

# Otros métodos

```
public static byte[] readAllBytes(Path path) throws IOException
// Lee todos los bytes y devuelve byte[].
public static Path write(Path path, byte[] bytes,
    OpenOption... options) throws IOException
// Opciones spn: CREATE, TRUNCATE_EXISTING, y WRITE

public static List<String> readAllLines(Path path, Charset cs)
    throws IOException
// Lee todas las líneas.
// Terminador de línea puede ser "\n", "\r\n" or "\r".
public static Path write(Path path, Iterable<? extends CharSequence> lines,
    Charset cs, OpenOption... options) throws IOException
```

# Buffered Character-based I/O para ficheros de texto

```
public static BufferedReader newBufferedReader(Path path,  
        Charset cs) throws IOException
```

```
public static BufferedWriter newBufferedWriter(Path path,  
        Charset cs, OpenOption... options) throws IOException
```

# Crear ficheros, directorios o enlaces

```
public static Path createFile(Path path, FileAttribute<?>... attrs)  
// Crea un nuevo fichero.
```

```
public static Path createDirectories(Path dir, FileAttribute<?>  
    ... attrs) throws IOException  
// Crea un nuevo directorio.
```

```
public static Path createSymbolicLink(Path link, Path target,  
    FileAttribute<?>... attrs) throws IOException  
// Crea un enlace simbólico
```

FIN

*Fin*