Tema 4

Entrada/Salida. Serialización de Objetos.

Departament d'Informàtica. Universitat de València

Índice

¿Por qué un tema sobre entrada/salida y serialización?

- La información puede estar almacenada en ficheros.

  - *Hay situaciones en las que los datos de entrada a una aplicación están en un fichero.*
  - *O situaciones en las que se desea guardar la información en un fichero para ser recuperada en otro instante.*

- La programación en red en Java se basa en operaciones de Entrada/Salida.

  - *La comunicación entre máquinas se realiza enviando o recibiendo información.*
  - *Estos intercambios se realizan utilizando operaciones de Entrada/Salida.*

- La serialización tiene que ver con la persistencia de objetos.

- *La transformación de un objeto (o conjunto de objetos) en una secuencia de bits se conoce como serialización. Se puede aplicar en dos situaciones.*
- *¿Qué ocurre con un objeto cuando finaliza la máquina virtual sobre la que existe? Se destruye. Se puede conseguir la persistencia de un objeto guardándolo para ser recuperado posteriormente.*
- *El mecanismo que rige el intercambio de objetos entre máquinas virtuales es la serialización.*

1. Entrada/Salida

I/O en Java no es más débil que la E/S en C o en C++, es simplemente diferente.

- *En C++ las clases de salida a pantalla y entrada de teclado están declaradas en el archivo de cabecera iostream.h. Las clases para el trabajo con ficheros están declaradas en el archivo de cabecera fstream.h*
- *La salida de un mensaje a pantalla se puede realizar utilizando el objeto cout de la clase ostream del siguiente modo:*

  cout << "El valor es:";
- *En java las clases de Entrada/Salida están definidas en el paquete java.io.*
- *La salida de un mensaje por pantalla en Java se puede realizar utilizando el atributo estático out definido dentro la clase* `System`*:*

  System.out.println("El valor es:");

Se asume que la ventana de instrucciones es la menos importante de las fuentes o destinos de E/S.

No hay más que ver lo que hay que hacer para leer una línea de texto.

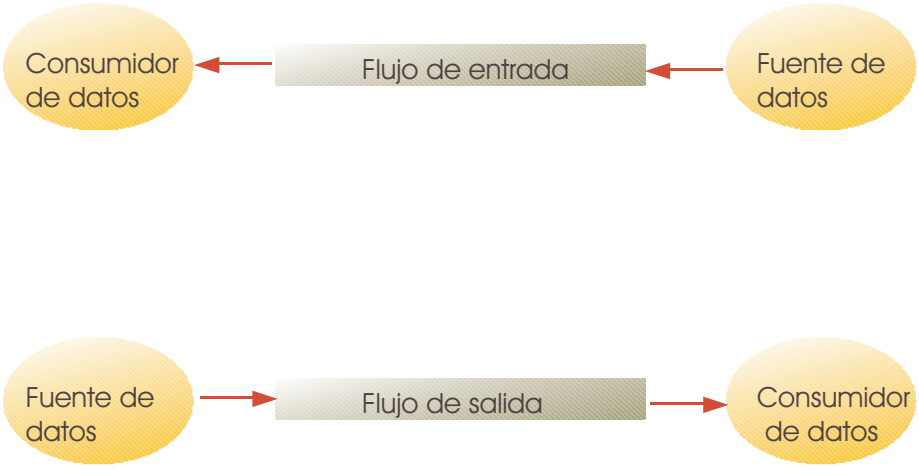
```
import java.io.*;

class EntradaConsola{

   public static void main(String[] args){
     try{
       System.out.println("Introduce texto");
       BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
       String cad = br.readLine();
       System.out.println(cad);
     }catch(IOException e){
       System.out.println("Error de E/S");
     }
```

```
    }
}
```

- La E/S en Java está basada en la noción de *streams* (flujos).

> *Un* **flujo** *es una abstracción de cualquier secuencia de datos yendo desde una fuente (o productor) hacia un destino (o consumidor). Se puede interpretar como una serie de datos circulando a través de un determinado canal que une a la fuente y al destino.*

- Los bytes se pueden procesar de uno en uno o en bloques (utilizando algún *buffer*)
- En algunos casos se conoce el número de bytes a leer (como en el caso de un fichero que tiene un tamaño) y en otros casos no se conoce el número de bytes a procesar (por ejemplo en una conexión de red)



Algunos ejemplos de fuentes o destinos de datos a los que se pueden asociar flujos de E/S:

- A la entrada estándar System.in, a la salida estándar System.out, a la salida de error estándar System.err
- Fichero binarios o ficheros de texto.
- Conexiones de red.
- Los propios programas: flujos de arrays de bytes, piped streams (para la comunicación entre hilos), etc.

Las clases **base** del paquete java.io son

Para flujos orientados a bytes:

- java.io.InputStream
- java.io.OutputStream

Para flujos orientados a caracteres:

- java.io.Reader
- java.io.Writer

Las cuatro clases anteriores están declaradas como abstractas.

1.1. Entrada orientada a bytes

- Los flujos de lectura extienden a la clase InputStream
- Esta clase define el método abstracto:
  public abstract int read() throws IOException
  Lee el siguiente byte y lo devuelve como un int en el rango 0 a 255.
  Si no hay más datos (porque se ha llegado al final) devuelve un -1 (este hecho se utiliza como condición de final).
- Este método bloquea la ejecución del programa esperando a que se de alguna de las circunstancias siguientes: el siguiente dato está disponible para ser leído, se detecta el final o se lanza una excepción.
- Cada subclase debe proporcionar una implementación de este método.
- El paquete java.io proporciona unos cuantos flujos de entrada y hay que saber cual es el apropiado para realizar de forma efectiva la lectura.

Tanto para la entrada y la salida orientadas a bytes como las orientadas a caracteres veremos que en la biblioteca java.io se ofrecen dos subconjuntos de clases:

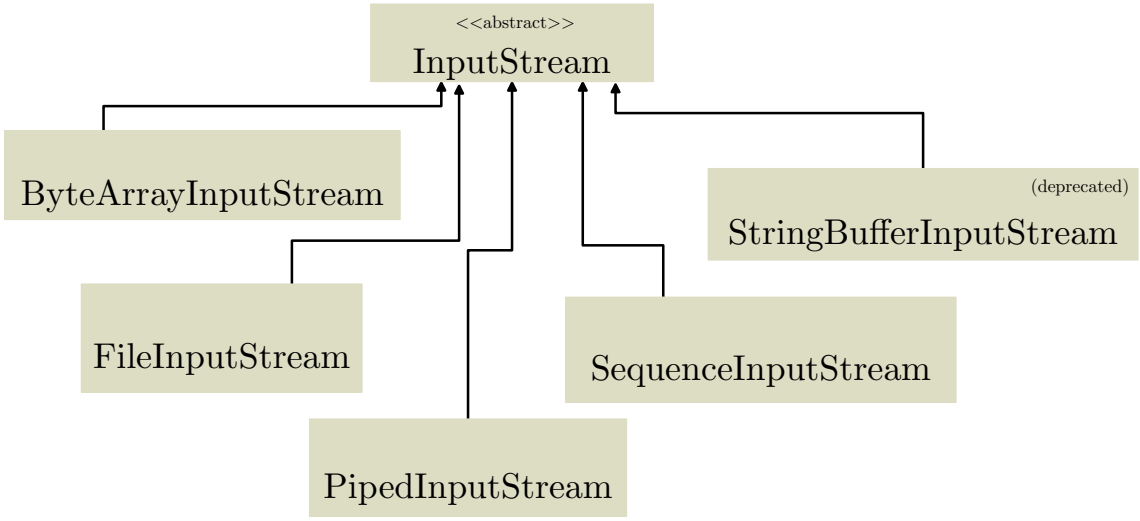
- *De bajo nivel* ofrece funcionalidad básica para la lectura y la escritura de bytes o de caracteres.
- *Filtrada* ofrece alguna funcionalidad añadida para la escritura y lectura de bytes o caracteres.

1.1.1. Entrada de bajo nivel orientada a bytes

La siguiente tabla muestra 5 clases que representan flujos de entrada de bajo nivel y que representan implementaciones a InputStream para diferentes propositos.

Flujo de entrada	Descripción
ByteArrayInputStream	permite leer bytes de un array que esté en memoria.
FileInputStream	permite leer bytes de un fichero del sistema local de ficheros.
PipedInputStream	permite leer bytes desde un *pipe* creado por un hilo.
StringBufferInputStream	permite leer lee bytes desde un String.
SequenceInputStream	permite leer bytes de datos desde dos o más flujos de bajo nivel, cambiando a otro flujo al finalizar la lectura de cada uno de ellos.

La jerarquía que siguen estas clases es:



Hay flujos de entrada que el usuario no puede crear de una forma directa sino que se obtienen a través de otros objetos.

Un ejemplo es el flujo de entrada que representa una conexión de red a través de un socket TCP, el proceso es el siguiente:

- Se crea el socket
- A través del socket se obtiene el flujo de entrada.

1.1.2. Utilización de un flujo de entrada de bajo nivel orientado a bytes

Listado 1: Ejemplo de entrada de bajo nivel orientada a bits

```
import java.io.*;

public class InputDemo{
   public static void main(String[] args){
     int dato;
     try{
       // Se supone que se pasa el nombre del fichero a
       // mostrar como un argumento al programa
       InputStream entrada = new FileInputStream( args[0]);

       while ( (dato=entrada.read())!=-1 )
         System.out.write( dato );

       System.out.println();

       entrada.close();
     }catch(IOException e){
       System.out.println("Error leyendo del fichero" + args[0]);
     }
```

```
    }
}
```

Esta no es la forma más eficiente de mostrar el contenido de un fichero pero sirve para ilustrar la utilización de los flujos de entrada de bajo nivel.

1.1.3. Entrada filtrada orientada a bytes

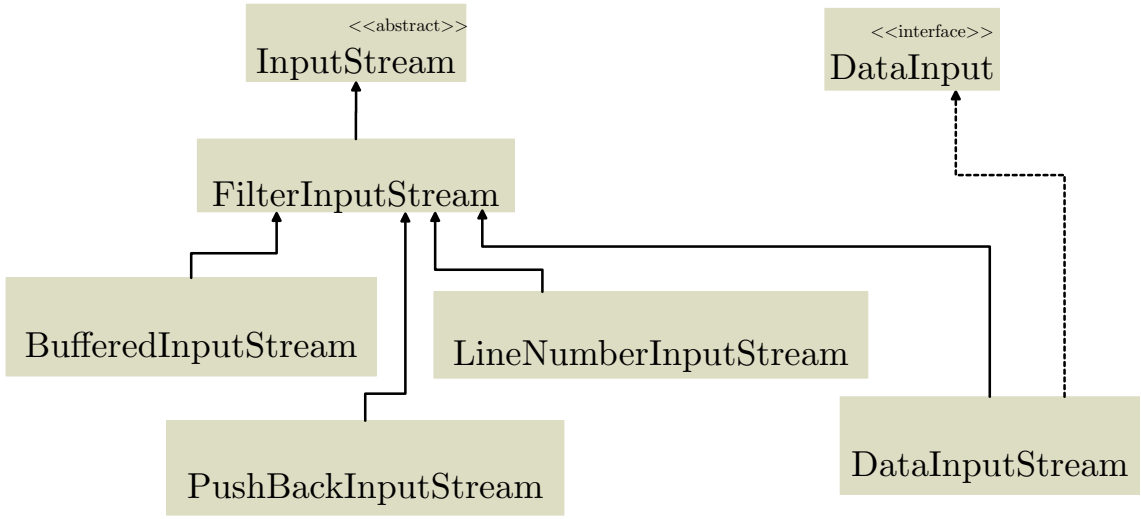
- A pesar de que los flujos de bajo nivel mostrados permiten leer bytes su flexibilidad es limitada.
- ¿Qué ocurre si deseamos leer una palabra? ¿o un entero? ¿o un carácter?
- Los flujos filtrados añaden funcionalidad a un flujo existente, procesando los datos de alguna forma (por ejemplo almacenando los datos en un buffer para mejorar la eficiencia) u ofreciendo métodos adicionales que permitan acceder a los datos de otros modos (por ejemplo leyendo una línea de texto en lugar de byte a byte).
- Los flujos filtrados de entrada extienden a la clase FilterInputStream
- Para crear un flujo filtrado hay que disponer en última instancia de un flujo de bajo nivel ya que hay que pasar mediante el constructor un flujo de bajo nivel existente.

Las clases que se definen en el paquete java.io para entrada mediante flujos filtrados son:

Flujo filtrado	Descripción
BufferedInputStream	almacena el acceso a los datos para mejorar la eficiencia.
DataInputStream	permite leer tipos de datos primitivos, tales como int, float, double e incluso líneas de texto.
LineNumberInputStream	mantiene un contador sobre el número de línea que se está leyendo, basado en una interpretación de los caracteres de final de línea.
PushBackInputStream	añade a otro flujo de entrada la capacidad de volver a colocar un byte leído (o array de bytes) otra vez en el flujo. La siguiente operación de lectura volverá a leer el último byte.

Otros paquetes definen más flujos de entrada filtrados (java.security, java.util, javax.crypto, java.util.zip).

La jerarquía que siguen estas clases es:



- Como se puede ver en el diagrama anterior, estas clases heredan de la clase FilterInputStream.
- El constructor de esta clase admite otro flujo de entrada, que es utilizado como fuente de datos (la lectura se realiza a través de éste), posiblemente transformando los datos conforme son leídos o proporcionando una funcionalidad adicional que no poseía el flujo original. El constructor es:

  ```
  protected FilterInputStream(InputStream in)
  ```

- La clase FilterInputStream simplemente oculta todos los métodos de InputStream con versiones que lo que realizan es pasar las peticiones al flujo de entrada con el que se creó.
- Las subclases de FilterInputStream pueden ocultar algunos de estos métodos y pueden proporcionar métodos o atributos adicionales.

Esta solución, se conoce como patrón **Decorator** y pertenece al grupo de **patrones estructurales**.

- *Un patrón de diseño es una solución que aparece frecuentemente en problemas de diseño de aplicaciones.*
- *Un patrón afronta un problema habitual de diseño que aparece en situaciones específicas y presenta una solución a él.*
- *Un patrón identifica y especifica abstracciones que están por encima del nivel de clases aisladas e instancias , o de componentes.*

The design Patterns Java Companion. Addison-Wesley, 1998

De estas definiciones se puede deducir que el proceso para encontrar patrones es un proceso inductivo: observando que las soluciones propuestas para una serie de problemas tienen una estructura similar.

1.1.4. Utilización de un flujo de entrada filtrado orientado a bytes

La clase FilterInputStream es un decorador con el que se puede envolver cualquier objeto que sea del tipo InputStream

Listado 2: Ejemplo de entrada filtrada orientada a bytes

```
import java.io.*;

class DemoDataInputStream{

   public static void main(String[] args){

     String s = "Esta es una demostracion de uso de DataInputStream";

     // Obtenemos una representacion en forma de bytes de esta cadena
     // Cada caracter esta representado por 1 byte
     byte[] bytes2 = s.getBytes();

     // Ahora creamos un array de bytes el doble de longitud
     byte[] bytes = new byte[2*s.length()];

     // Cada byte que se obtuvo con getBytes se convierte en dos
     // el primero se pone al cero y el segundo se deja con el
     // valor que se obtuvo con getBytes
     int cont=0;
     for (int i=0;i<2*s.length();i=i+2){
       bytes[i]=0;
       bytes[i+1]=bytes2[cont];
       cont++;
     }
```

```
     // A partir del array de bytes se obtiene un ByteArrayInputStream
     // a partir de este un BufferedInputStream y a partir de este un
     // DataInputStream
     DataInputStream in = new DataInputStream(
         new BufferedInputStream(new ByteArrayInputStream(bytes)));


     try{
       while (true){
         System.out.print(in.readChar());
       }
     // Cuando no hayan mas datos se lanzara esta excepcion
     }catch(EOFException e){
       System.out.println();
       System.out.println("Final de lectura");
     }catch(IOException e){
       System.out.println("Error de lectura");
     }
   }
}
```

1.2. Salida orientada a bytes

- Los flujos de escritura extienden a la clase OutputStream
- Esta clase define el método abstracto:

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

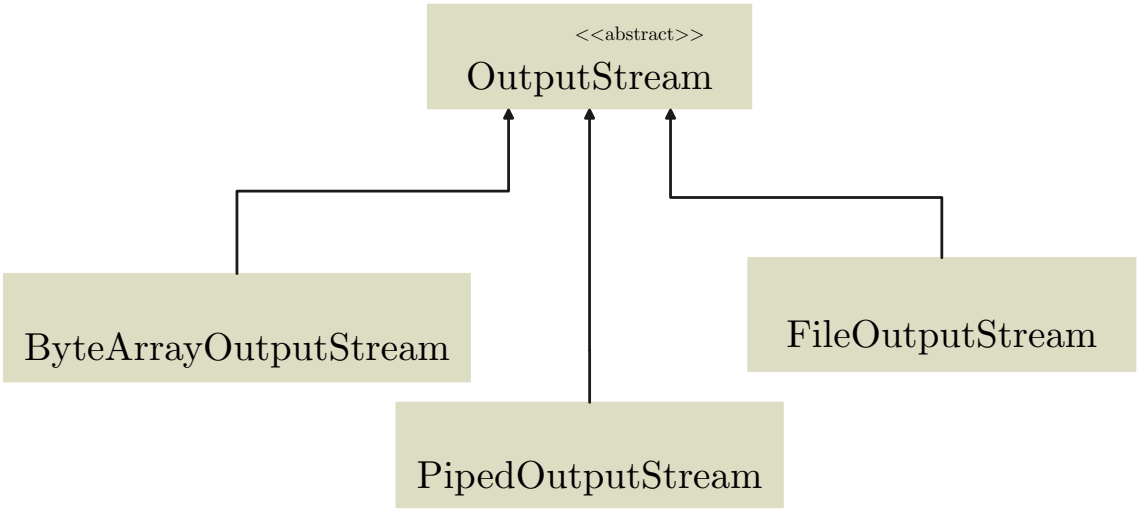
  Escribe un byte en este flujo de salida. El byte que se escribe se obtiene como los 8 bits de más bajo orden del argumento b. Los restantes bits (24) de b se desechan.
- Cada subclase debe proporcionar una implementación de este método.
- El paquete java.io proporciona unos cuantos flujos de salida y hay que saber cual es el apropiado para realizar de forma efectiva la escritura.

1.2.1. Salida de bajo nivel orientada a bytes

La siguiente tabla muestra 3 clases que representan flujos de salida de bajo nivel.

Flujo de entrada	Descripción
ByteArrayOutputStream	escribe bytes de datos a un array que esté en memoria.
FileOutputStream	escribe bytes de datos a un fichero del sistema local de ficheros.
PipedOutputStream	escribe bytes de datos a un pipe creado por un hilo.

La jerarquía que siguen estas clases es:



1.2.2. Utilización de un flujo de salida de bajo nivel orientado a bytes

Listado 3: Ejemplo de salida de bajo nivel orientada a bytes

```
import java.io.*;

public class OutputDemo{
   public static void main(String[] args){
     int dato;
     try{
       // Se supone que se pasa el nombre del fichero a
       // crear como un argumento al programa
       OutputStream salida = new FileOutputStream( args[0]);

       String s = "Esta es una cadena de prueba";

       byte[] bytes = s.getBytes();

       for (int i=0; i<bytes.length; i++)
         salida.write(bytes[i]);

       salida.close();
```

```
      }catch(IOException e){
        System.out.println("Error escribiendo en el fichero" + args[0]);
      }
    }
}
```

1.2.3. Salida filtrada orientada a bytes

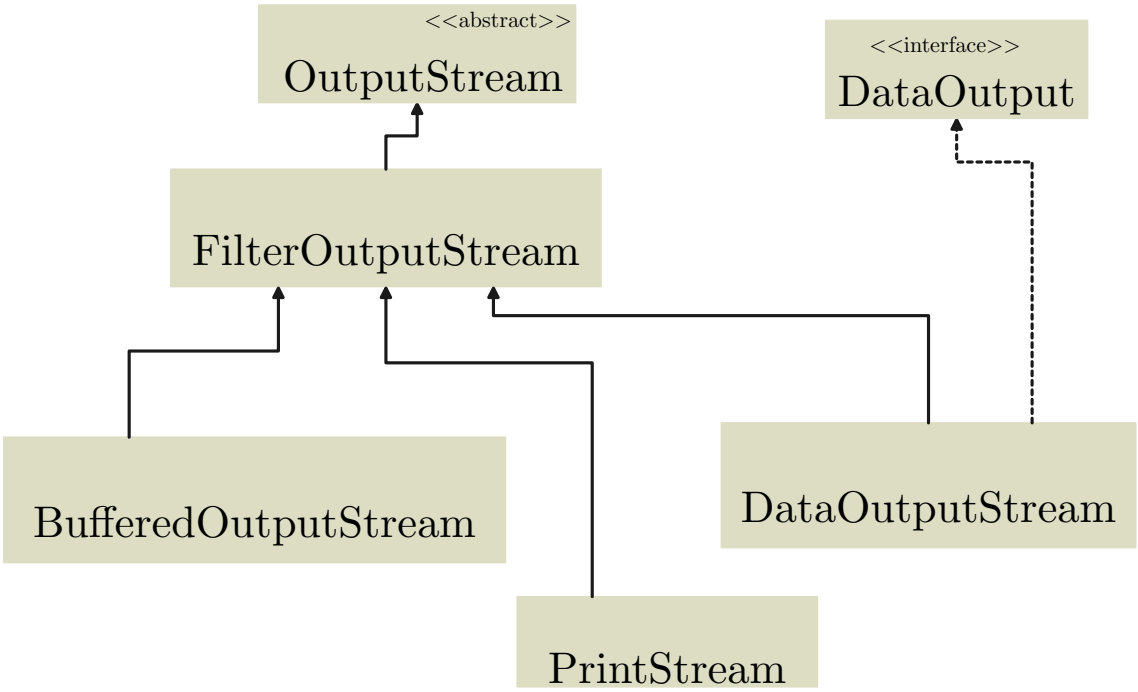
Las clases de salida también utilizan el patrón decorador mediante la clase FilterOutputStream y sus subclases.

El objetivo es el mismo que anteriormente: añadir una determinada funcionalidad.

Las clases que se definen en el paquete java.io para salida mediante flujos filtrados se muestran en la siguiente tabla.

Flujo filtrado	Descripción
BufferedOutputStream	Esta clase implementa un flujo de salida con buffer, de forma que una aplicación puede escribir bytes en el flujo de salida subyacente sin que necesariamente se produzcan llamadas para que se escriba cada byte (se pueden almacenar para ser escritos en bloque).
DataOutputStream	Permite a las aplicaciones escribir tipos Java primitivos en un flujo de salida de un modo portable. Se puede utilizar un DataInputStream para leer los datos.
PrintStream	Añade la capacidad de escribir representaciones de diversos datos de forma conveniente. Los métodos de esta clase no lanzan una excepción del tipo IOException, en su lugar establecen una bandera interna que puede ser consultada mediante el método checkError. La salida estándar (a la que se puede acceder a través del atributo estático out de la clases System es de este tipo.

La jerarquía de estas clases es



1.2.4. Utilización de un flujo de salida filtrado orientado a bytes

Listado 4: Ejemplo de salida filtrada orientada a bytes

```
import java.io.*;

class DemoDataOutputInputStream{

   public static void main(String[] args){

     // Primero se escriben datos de diverso tipo a un fichero
     try{
       DataOutputStream out = new DataOutputStream(
         new FileOutputStream("borrame.bin"));
       out.writeChar('A');
       out.writeBoolean(true);
       out.writeInt(256);
       out.close();
     }catch(IOException e){};

     // A continuación se leen del fichero y se muestran
     try{
       DataInputStream in = new DataInputStream(
         new FileInputStream("borrame.bin"));
       System.out.println(in.readChar());
       System.out.println(in.readBoolean());
       System.out.println(in.readInt());
       in.close();
     }catch(IOException e){};

   }
}
```

1.3. Entrada orientada a caracteres

- Los flujos vistos anteriormente se pueden utilizar para leer y escribir texto además de bytes y tipos primitivos de datos.
- Sin embargo, para trabajar con texto es mejor utilizar las clases derivadas de Reader y Writer
- Estas clases fueron introducidas a partir de la versión 1.1 del JDK para dar soporte a flujos de caracteres Unicode.
- La clase Reader tiene los mismos métodos que InputStream salvo que los distintos métodos read() trabajan con 2 bytes en lugar de con byte.
- La clase Writer tiene los mismos métodos que OutputStream salvo que los distintos métodos write() trabajan como en el caso anterior con 2 bytes.

1.3.1. ¿Qué son los caracteres Unicode?

Unicode es un conjunto de caracteres extendido.

El código ASCII representa un conjunto de caracteres utilizando un byte (lo cual arroja la cantidad de 256 símbolos), lo cual es insuficiente para representar los muchos caracteres existentes.

Los caracteres Unicode están representados mediante uno o más bytes (dependiendo de la codificación su longitud puede variar entre uno y cuatro bytes).

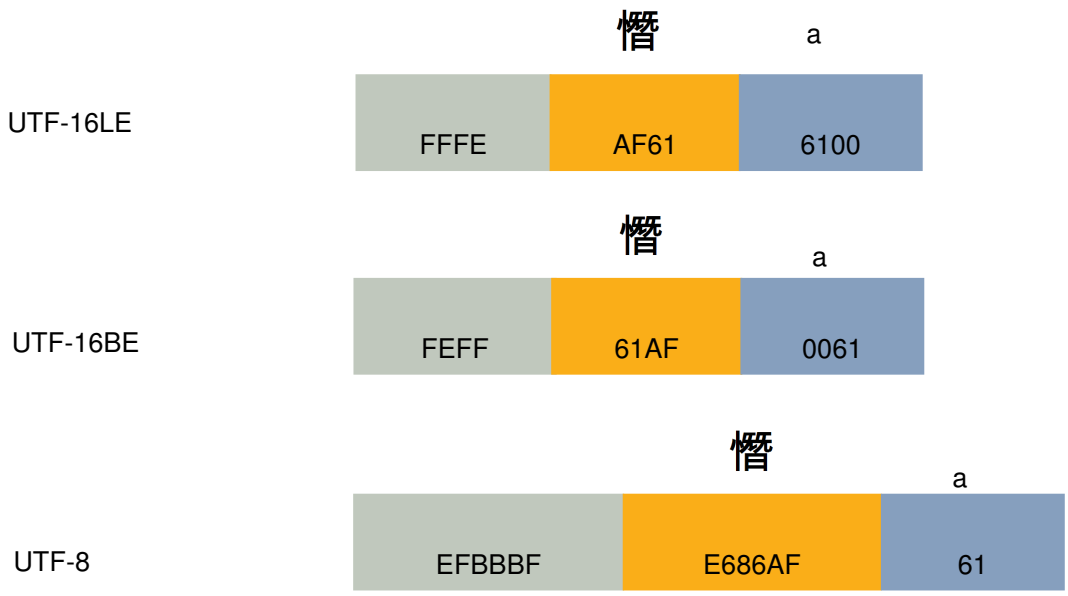
Java ofrece soporte para el trabajo con caracteres Unicode. Además ofrece soporte para una variante de Unicode llamada UTF-8 (una forma de codificación en la que a cada carácter se le asigna una secuencia de uno a cuatro bytes sin signo).

La siguiente tabla muestra el conjunto mínimo de codificaciones que debe estar soportado por toda máquina virtual de Java.

Flujo de entrada	Descripción
US-ASCII	código ASCII de 7 bits, ISO646-US, es el bloque básico de Latin del conjunto de caracteres Unicode
ISO-8859-1	ISO-LATIN-1
UTF-8	Unicode Transformation Format 8; es similar al US-ASCII (0x00..0x7F). Esto significa que en este formato estos caracteres tienen un único byte que coincide con el del código US-ASCII. Para el resto de los caracteres utiliza más bytes. Esta es la forma de codificación preferida para HTML otros protocolos similares, particularmente para Internet ya que su transparencia con ASCII facilita la migración.
UTF-16BE	Unicode Transformation Format de 16 bits; con un orden de bytes big-endian
UTF-16LE	Unicode Transformation Format de 16 bits; con un orden de bytes low-endian
UTF-16	Unicode Transformation Format de 16 bits, en el que el orden de los bytes está especificada por una marca inicial que indica el orden.

Algunas máquinas virtuales ofrecen soporte para otros conjuntos de caracteres (por ejemplo en Windows se ofrece soporte para Cp1252 que es una modificación del ASCII).

La siguiente ilustración muestra los bytes (en hexadecimal) de un fichero con dos caracteres para diferentes formatos.



1.3.2. Entrada de bajo nivel orientada a caracteres

- La clase de la que heredan las clases de entrada que trabajan con caracteres es la clase `Reader` que está definida como abstracta.

- Una clase que la extienda debe implementar los métodos read(char[], int, int) y close(). Pero además puede ocultar (o sobreescribir) alguno de los demás métodos para proporcionar mayor eficiencia, funcionalidad adicional o ambos.

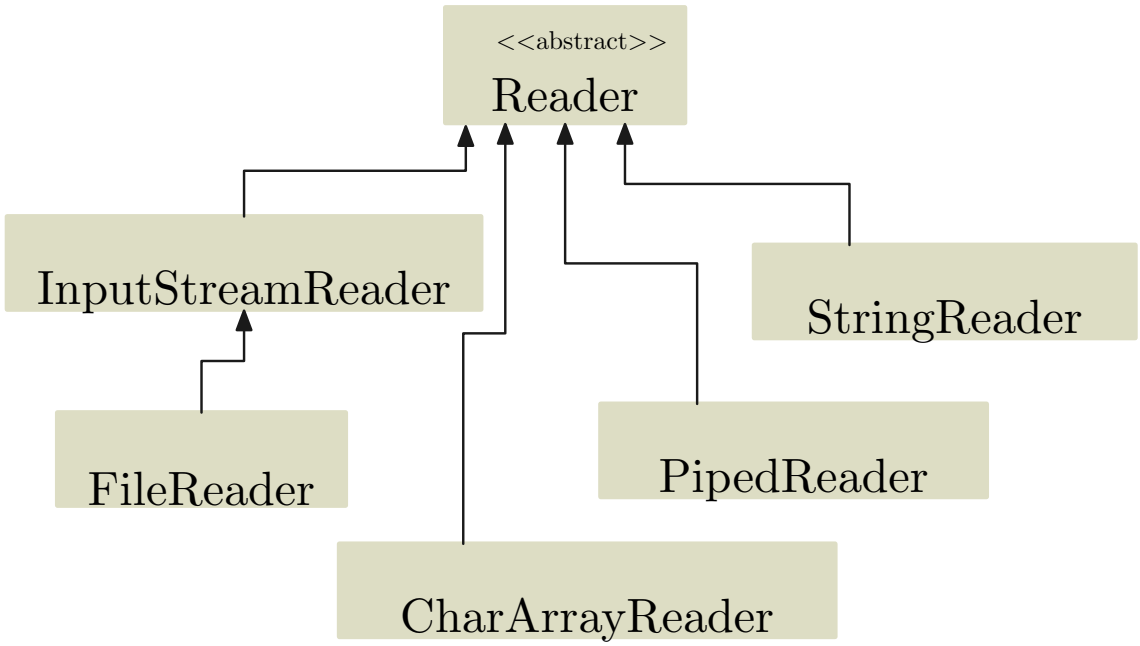
- Tiene el método (entre otros)
  public int read() throws IOException
  para leer un único carácter devuelto como un entero en el rango 0 a 65535 (0x0 - 0xFFFF) o -1 si se ha llegado al final del fichero.

- Este método bloqueará hasta que el siguiente carácter esté disponible, se lance una excepción o se alcance el final del fichero.

Las clases que extienden a Reader para la lectura de caracteres de bajo nivel en el paquete `java.io` son:

Flujo de entrada	Descripción
CharArrayReader	Lee de un array de caracteres.
FileReader	Lee de un fichero en el sistema de ficheros local.
PipedReader	Lee caracteres de un pipe de comunicación entre hilos.
StringReader	Lee caracteres de un String.
InputStreamReader	Permite asociar un *reader* a un *input stream* leyendo del último.

La siguiente figura muestra la jerarquía de las clases anteriores.



1.3.3. Utilización de un flujo de entrada de bajo nivel orientado a caracteres

En este ejemplo se lee un fichero de caracteres asumiendo la codificación de caracteres por defecto.

Listado 5: Ejemplo de entrada de bajo nivel orientada a caracteres (1)

```
import java.io.*;

class DemoFileReader{

   public static void main(String[] args){
     int c;

     try{
       FileReader fr = new FileReader(args[0]);

       while ((c=fr.read())!=-1)
         System.out.print((char)c);
       fr.close();
     }catch(IOException e){
       System.out.println("Error E/S");
       e.printStackTrace();
     }

   }
}
```

Si se ejecuta en Windows utilizará la codificación por defecto que es la Cp1252.

Si se le pasa un fichero con cualquier formato Unicode no funcionará correctamente.

En este otro ejemplo se lee un fichero de caracteres proporcionando la codificación.

Listado 6: Ejemplo de entrada de bajo nivel orientada a caracteres (2)

```
import java.io.*;

class DemoInputStreamReaderCodificacion{

   public static void main(String[] args){
     int c;
```

```
      try{
        InputStreamReader fr = new InputStreamReader(new
            FileInputStream(args[0]),"UTF-8");

        while ((c=fr.read())!=-1)
          System.out.print((char)c);
        fr.close();
      }catch(IOException e){
        System.out.println("Error E/S");
        e.printStackTrace();
      }
    }
}
```

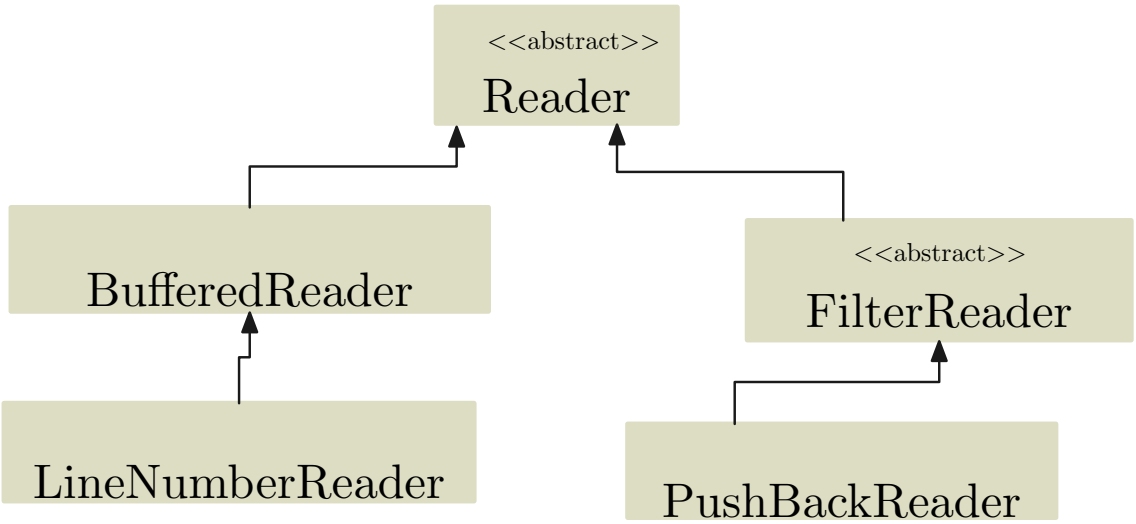
Nota: este programa solo mostrará los caracteres que coincidan con ASCII ya que la consola no soporta Unicode.

1.3.4. Entrada filtrada orientada a caracteres

Las clases que se ofrecen para entrada filtrada son:

Flujo de entrada	Descripción
BufferedReader	Lee caracteres de un fichero, almacenando los caracteres para que la lectura sea eficiente.
PuschBackReader	Permite volver a poner en el flujo caracteres leídos.
LineNumberReader	Lee caracteres de un fichero utilizando almacenamiento y permitiendo consultar el número de líneas leídas.

La jerarquía que siguen estas clases es:



1.3.5. Utilización de un flujo de entrada filtrado orientado a caracteres

Este ejemplo es una modificación del listado 6.

Listado 7: Ejemplo de entrada filtrada orientada a caracteres

```
import java.io.*;

class DemoBufferedReader{

   public static void main(String[] args){
     String cad;

     try{
       BufferedReader br = new BufferedReader(
          new InputStreamReader(
            new FileInputStream(args[0]),"UTF-8"));

       while ( (cad=br.readLine())!=null )
         System.out.println(cad);
       br.close();
     }catch(IOException e){
       System.out.println("Error E/S");
       e.printStackTrace();
     }
   }
}
```

Nota: este programa solo mostrará los caracteres que coincidan con ASCII ya que la consola no soporta Unicode.

1.4. Salida orientada a caracteres

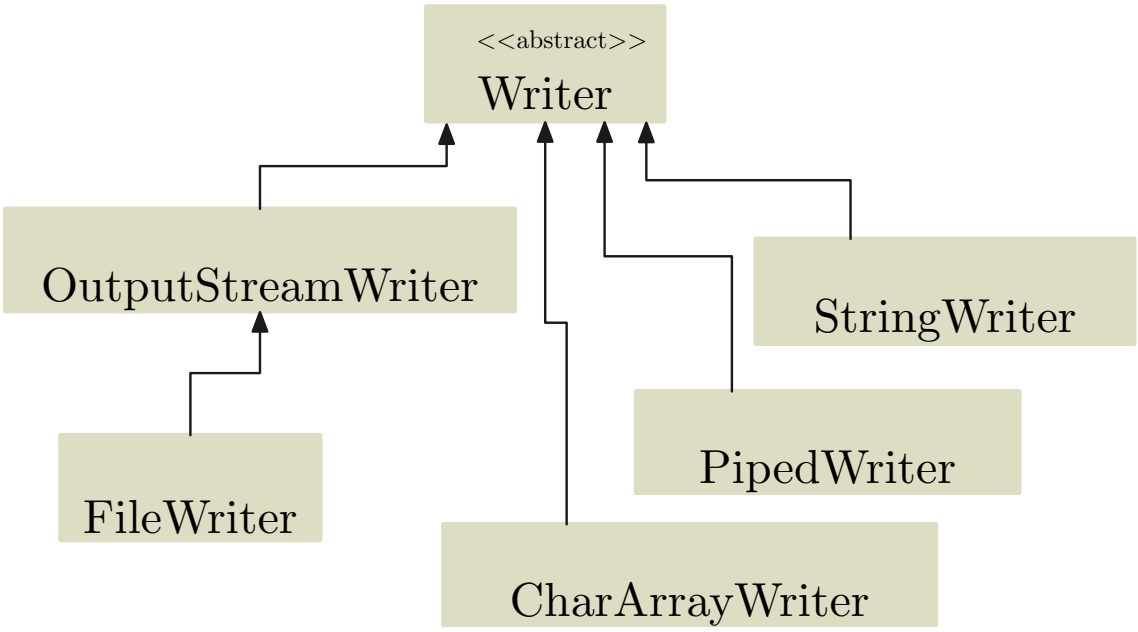
- La clase de la que heredan las clases de salida que trabajan con caracteres es la clase `Writer` que está definida como abstracta.
- Una clase que la extienda debe implementar los métodos write(char[], int, int), flush() y close(). Pero además puede ocultar (o sobreescribir) alguno de los demás métodos para proporcionar mayor eficiencia, funcionalidad adicional o ambos.
- Tiene el método (entre otros)
  public write(int) throws IOException
  para escribir un único carácter.
- El caracter a escribir está contenido en los 16 bits de bajo orden del valor entero proporcionado, los 16 bits de más alto orden son desechados.

1.4.1. Salida de bajo nivel orientada a caracteres

Las clases que extienden a Reader para la lectura de caracteres de bajo nivel en el paquete `java.io` son:

Flujo de entrada	Descripción
CharArrayWriter	Escribe en un array de caracteres.
FileWriter	Escribe en un fichero en el sistema de ficheros local.
PipedWriter	Escribe caracteres de un pipe de comunicación entre hilos.
StringWriter	Escribe caracteres en un String.
OutputStreamWriter	Permite asociar un *writer* a un *output stream* escribiendo en el último.

La jerarquía que siguen estas clases es:



1.4.2. Utilización de un flujo de salida de bajo nivel orientado a caracteres

Listado 8: Ejemplo de salida de bajo nivel orientada a caracteres

```
import java.io.*;

class Productor extends Thread{
   private PipedWriter out;
   private String texto;

   Productor(PipedWriter p, String txt){
     out = p;
     texto = txt;
   }

   public void run(){
     try{

       StringReader sr = new StringReader(texto);
       int c;
       while ((c=sr.read())!=-1)
         out.write(c);
       out.flush();
       out.close();
     }catch(IOException e){
       System.out.println("Error de escritura");
     }

   }
}
```

```
class Consumidor extends Thread{
   private PipedReader in;
   private int dif;

   Consumidor(PipedReader p,int d){
     in = p;
     dif = d;
   }

   public void run(){
     int c;
     try{
       while ((c = in.read())!=-1)
         System.out.print((char)(c+dif));
     }catch(IOException e){
       System.out.println("Error de lectura");
     }
   }
}
```

```
class DemoPipes{

   public static void main(String[] args){
     try{
       PipedWriter pw = new PipedWriter();
       PipedReader pr = new PipedReader(pw);
       Productor prod = new Productor(pw,"Texto a enviar al consumidor");
       Consumidor cons = new Consumidor(pr,Integer.parseInt(args[0]));
       prod.start();
       cons.start();

      try{
         prod.join();
         cons.join();
       }catch(InterruptedException e){}

       System.out.println("\nHilos finalizados");
     }catch(IOException e){}

   }
}
```

Salida pasando como argumento 0:

```
Texto a enviar al consumidor
Hilos finalizados
```

Salida pasando como argumento −10:

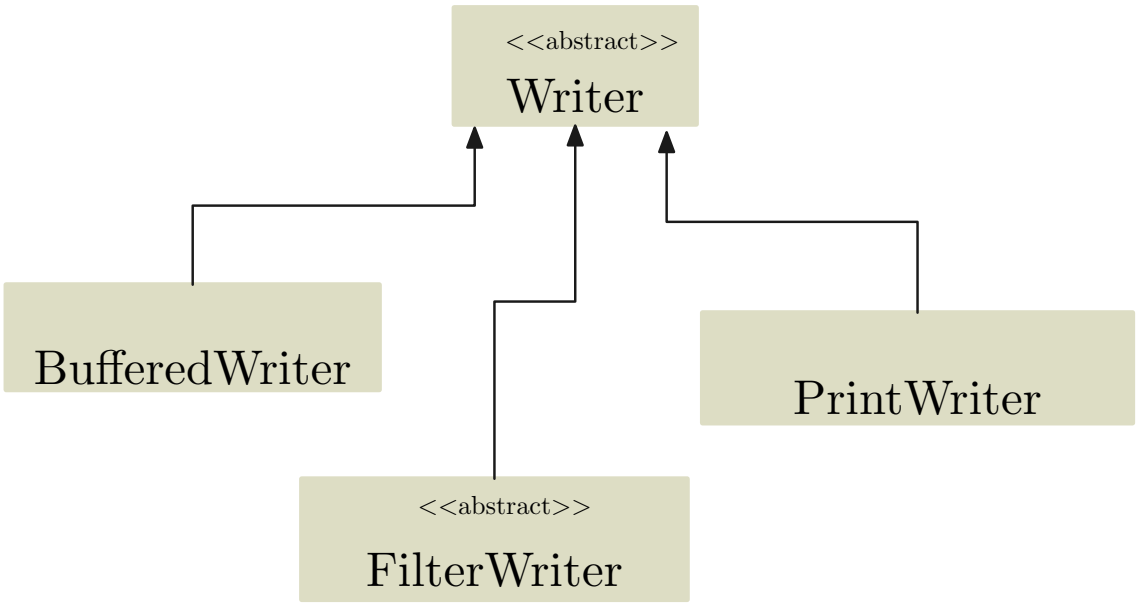
```
J[nje?W?[dl_Wh?Wb?Yedikc_Zeh
Hilos finalizados
```

1.4.3. Salida filtrada orientada a caracteres

Las clases que se ofrecen para salida filtrada son:

Flujo de entrada	Descripción
BufferedWriter	Lee caracteres de un fichero, almacenando los caracteres para que la lectura sea eficiente.
PrintWriter	Permite escribir tipos primitivos y objetos a un flujo de salida orientado a caracteres.

La jerarquía que siguen estas clases es:



1.4.4. Utilización de un flujo de salida filtrado orientado a caracteres

Listado 9: Ejemplo de salida filtrada orientada a caracteres

```
import java.io.*;
import java.util.*;
class Punto{
   private int x;
   private int y;
   Punto(int cx, int cy)
     x = cx;
     y = cy;
   }
   public String toString(){
     return "(" + x + ", " + y + ")";
   }
}
class DemoPrintWriter{
   public static void main(String[] args){
     try{
       int i=20;
       LinkedList lista = new LinkedList();
       lista.add(new Punto(0,0));
       lista.add(new Punto(1,1));
       lista.add(new Punto(2,2));

       PrintWriter pw = new PrintWriter(
         new BufferedOutputStream(new FileOutputStream("fich.txt")),true);

       pw.println(lista);
       pw.println(i);
       pw.close();
     }catch(IOException e){}
   }
}
```

Ejecutando el programa anterior se crea el fichero `fich.txt` cuyo contenido es:

```
[(0, 0), (1, 1), (2, 2)]
20
```

La información sobre los puntos se obtiene del método toString() de la clase Punto.

2. Serialización

2.1. ¿Qué es la serialización?

- La **serialización** de objetos permite convertir cualquier objeto que implemente a la interfaz Serializable o la interfaz Externalizable en una secuencia de bits que puede ser utilizada posteriormente para reconstruir el objeto original.

- Esta secuencia de bits puede guardarse en un fichero o puede enviarse a otra máquina virtual (que puede estar ejecutándose en otro sistema operativo) para reconstruir el objeto en otro instante o en otra máquina virtual.

- La posibilidad de guardar un objeto de forma que pueda existir incluso cuando la aplicación haya finalizado se conoce como **persistencia**.

- Los objetos mantienen referencias a otros objetos. Estos otros objetos deben ser también almacenados y recuperados con el fin de mantener las relaciones originales. Por supuesto, todos estos objetos deben ser serializables ya que de lo contrario se lanzará una excepción del tipo NotSerializableException.

- Para reconstruir un objeto (o conjunto de objetos) Java serializado es necesario que la clase (o clases) esté en el `classpath` con el fin de indentificarla y verificarla antes de restaurar el contenido en una nueva instancia.

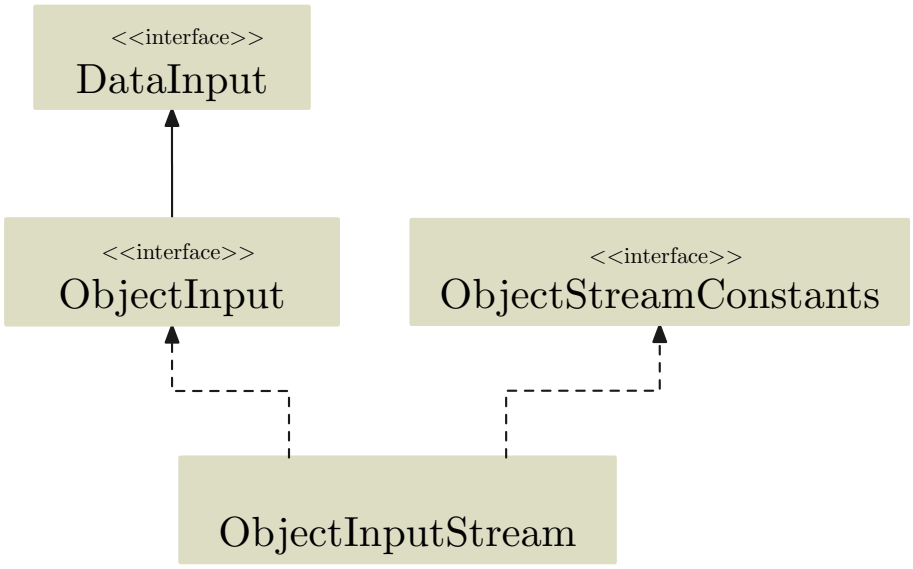
- La interfaz Serializable no define ningún método sirve como un indicador.

- La serialización se introdujo en Java para soportar la Invocación Remota de Métodos (RMI) que permite a una aplicación enviar mensajes a un objeto remoto (que se esté ejecutando en otra máquina virtual). También es necesaria en el caso de los JavaBeans.

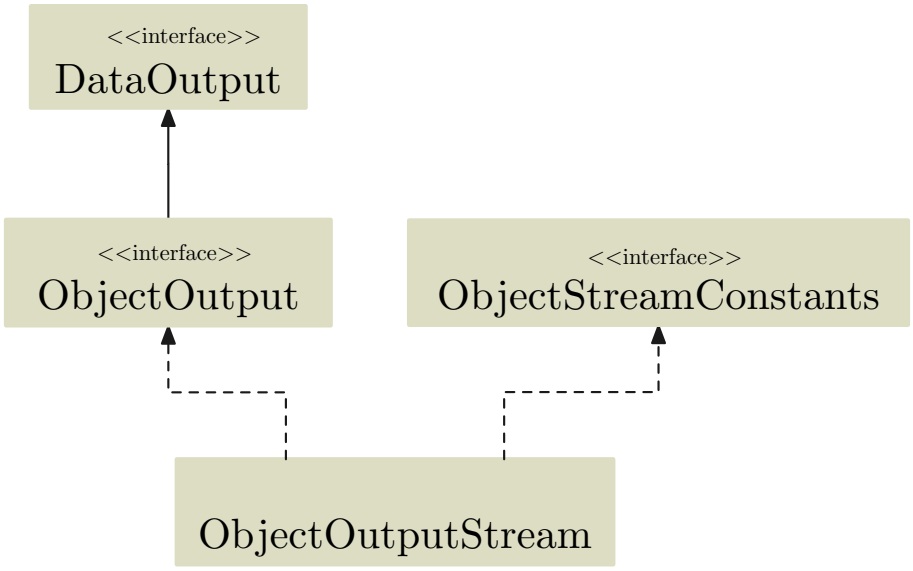
2.1.1. Flujos para entrada y salida de objetos

- Java proporciona clases para crear flujos de entrada y de salida de objetos.
- La serialización está orientada a bytes por lo tanto se utilizan clases que estén en la jerarquía de InputStream u OutputStream.
- Para serializar un objeto es necesario crear algún objeto del tipo OutputStream que se le pasará al constructor de ObjectOutputStream. A continuación se puede llamar a writeObject() para serializar el objeto.
- Para recuperar un objeto es necesario crear algún objeto del tipo InputStream que se le pasará al constructor de ObjectInputStream. A continuación se puede llamar a readObject() para leer el objeto.

La jerarquía que siguen las clases para la recuperación de objetos serializados es:



La jerarquía que siguen las clases para la serialización de objetos es:



Ejemplo. En este ejemplo hay 3 clases. Una clase Punto y ListaPuntos que implementan a Serializable y otra clase DemoSerializacion que contiene el main. En el main se crea un objeto de la clase ListaPuntos y se almacena en un fichero, a continuación se lee el objeto y se le envían mensajes para comprobar que se ha reconstruido correctamente.

Listado 10: Ejemplo de serialización de objetos

```
import java.io.*;
import java.util.*;

/** Clase que representa un punto 2d*/
class Punto implements Serializable{
    private int x;
    private int y;

    Punto(int cx, int cy){
      System.out.println("Creando el punto (" + cx + ", " + cy + ")");
      x = cx;
      y = cy;
    }

    public int getX(){
      return x;
    }

    public int getY(){
      return y;
    }
}
```

```
class ListaPuntos implements Serializable{
    private LinkedList lista = new LinkedList();

    ListaPuntos(int nPuntos){
      System.out.println("Constructor de ListaPuntos");
      int x,y;

      for (int i=0; i<nPuntos; i++){
        x = (int) (Math.random()*10);
        y = (int) (Math.random()*10);
        lista.add(new Punto(x,y));
      }
    }

    public void muestraPuntos(){
      ListIterator li = lista.listIterator(0);
      Punto p;

      while (li.hasNext()){
        p=(Punto)li.next();
        System.out.println("x = " + p.getX() + ", y = " + p.getY());
      }
```

```
    }
}
```

```
public class DemoSerializacion{
   public static void main(String[] args){
     ListaPuntos s = new ListaPuntos(5);

     try{
       ObjectOutputStream salida = new ObjectOutputStream(
         new FileOutputStream("objeto.bin"));

       salida.writeObject(s);

       salida.close();

       System.out.println("Recuperando el objeto...");

       ObjectInputStream entrada = new ObjectInputStream(
         new FileInputStream("objeto.bin"));

       ListaPuntos lp = (ListaPuntos)entrada.readObject();

       entrada.close();

       lp.muestraPuntos();
     }catch(FileNotFoundException fnfe){ System.out.println("Error E/S");
     }catch(IOException ioe){ System.out.println("Error E/S");
     }catch(ClassNotFoundException ioe){ System.out.println("Clase no encontrada");
     }
   }
}
```

La ejecución del programa anterior produce el siguiente resultado:

```
Constructor de ListaPuntos
Creando el punto (7, 4)
Creando el punto (8, 7)
Creando el punto (3, 2)
Creando el punto (0, 3)
Creando el punto (6, 4)
Recuperando el objeto...
x = 7, y = 4
x = 8, y = 7
x = 3, y = 2
x = 0, y = 3
x = 6, y = 4
Press any key to continue...
```

Como puede observarse, para crear la instancia a partir del objeto almacenado en el fichero no se llama a ninguno de los constructores, se crea a partir de los bytes almacenados.

- Si un atributo no se desea serializar, se puede declarar como `transient`.
- Otra opción a la hora de controlar qué es lo que se serializa de un objeto es implementar la interfaz Externalizable en lugar de la interfaz Serializable.
- La interfaz Externalizable extiende a Serializable añadiendo dos métodos, writeExternal(·) y readExternal(·) que son llamados automáticamente durante la serialización y la recuperación.
- Si un objeto implementa a Externalizable no se serializa automáticamente nada y se debe especificar lo que se debe serializar mediante llamadas a writeExternal().

Listado 11: Ejemplo de serialización y externalización de objetos

```
import java.io.*;
import java.util.*;

class Usuario implements Externalizable{
```

```
    private String usuario;
    private String password;

    public Usuario(){
      System.out.println("Creando usuario vacio");
    }

    Usuario(String u, String p){
      System.out.println("Creando Usuario (" + u + ", " + p + ")");
      usuario = u;
      password = p;
    }

    public void writeExternal(ObjectOutput out)
        throws IOException {
      System.out.println("Usuario.writeExternal");
      // Explicitamente indicamos cuales son los atributos a almacenar
      out.writeObject(usuario);
    }

    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
      System.out.println("Usuario.readExternal");
      // Explicitamente indicamos cuales son los atributos a recuperar
      usuario = (String)in.readObject();
    }
```

```
    public void muestraUsuario(){
      String cad="Usuario: " + usuario + "    Password: ";
      if (password == null)
        cad = cad + "No disponible";
      else
        cad = cad + password;
      System.out.println(cad);
    }
}

class ListaUsuarios implements Serializable{
    private LinkedList lista = new LinkedList();
    int valor;

    ListaUsuarios(String[] usuarios, String[] passwords){

      for (int i=0; i<usuarios.length; i++)
        lista.add(new Usuario(usuarios[i],passwords[i]));
    }

    public void muestraUsuarios(){
      ListIterator li = lista.listIterator(0);
      Usuario u;

      while (li.hasNext()){
        u=(Usuario)li.next();
        u.muestraUsuario();
      }
    }
}
```

```
class DemoExternalizable{
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {

      System.out.println("Creando el objeto");

      String[] usuarios={"A","B","C"};
      String[] passwords={"1","2","3"};

      ListaUsuarios lp = new ListaUsuarios(usuarios,passwords);

      System.out.println("\nAlmacenando objeto");
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("objetos.out"));

      o.writeObject(lp);
      o.close();

      System.out.println("\nRecuperando objeto");
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("objetos.out"));
      lp = (ListaUsuarios)in.readObject();
      lp.muestraUsuarios();
    }
}
```

La ejecución del programa anterior produce el siguiente resultado:

```
Creando el objeto
Creando Usuario (A, 1)
Creando Usuario (B, 2)
Creando Usuario (C, 3)

Almacenando objeto
Usuario.writeExternal
Usuario.writeExternal
Usuario.writeExternal

Recuperando objeto
Creando usuario vacio
Usuario.readExternal
Creando usuario vacio
Usuario.readExternal
Creando usuario vacio
Usuario.readExternal
Usuario: A    Password: No disponible
Usuario: B    Password: No disponible
Usuario: C    Password: No disponible
Press any key to continue...
```

Como puede observarse, (al contrario que en la serialización) al recuperar un objeto que ha sido *externalizado* se llama al constructor por defecto así que este debe ser accesible.

3. Índice de Listados de Código

Índice de listados de código