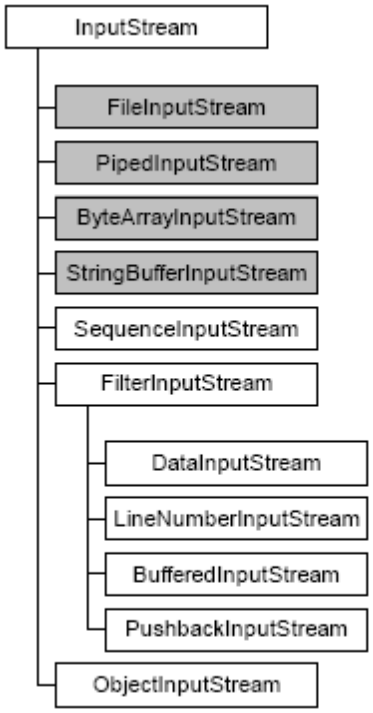
ENTRADA/SALIDA DE DATOS EN JAVA 1.1

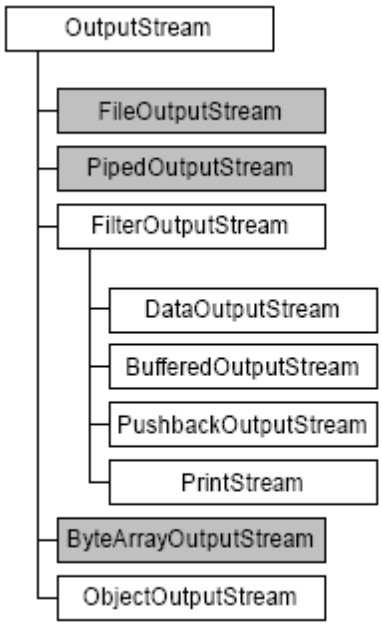
Los programas necesitan comunicarse con su entorno, tanto para recoger datos e información que deben procesar, como para devolver los resultados obtenidos.
La manera de representar estas entradas y salidas en ***Java*** es a base de ***streams*** (flujos de datos). Un ***stream*** es una conexión entre el programa y la fuente o destino de los datos. La información se traslada ***en serie*** (un carácter a continuación de otro) a través de esta conexión. Esto da lugar a una forma general de representar muchos tipos de comunicaciones.
Por ejemplo, cuando se quiere imprimir algo en pantalla, se hace a través de un ***stream*** que conecta el monitor al programa. Se da a ese ***stream*** la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/escritura de archivos, la comunicación a través de Internet o la lectura de la información de un sensor a través del puerto en serie.

1 CLASES DE JAVA PARA LECTURA Y ESCRITURA DE DATOS

El package ***java.io*** contiene las clases necesarias para la comunicación del programa con el exterior. Dentro de este package existen dos familias de jerarquías distintas para la entrada/salida de datos.
La diferencia principal consiste en que una opera con ***bytes*** y la otra con ***caracteres*** (el carácter de ***Java*** está formado por dos bytes porque sigue el código ***Unicode***). En general, para el mismo fin hay dos clases que manejan bytes (una clase de entrada y otra de salida) y otras dos que manejan caracteres.
Desde ***Java 1.0***, la entrada y salida de datos del programa se podía hacer con clases derivadas de ***InputStream*** (para lectura) y ***OutputStream*** (para escritura). Estas clases tienen los métodos básicos ***read()*** y ***write()*** que manejan ***bytes*** y que no se suelen utilizar directamente. La Figura 1 muestra las clases que derivan de ***InputStream*** y la Figura 2 las que derivan de ***OutputStream***.

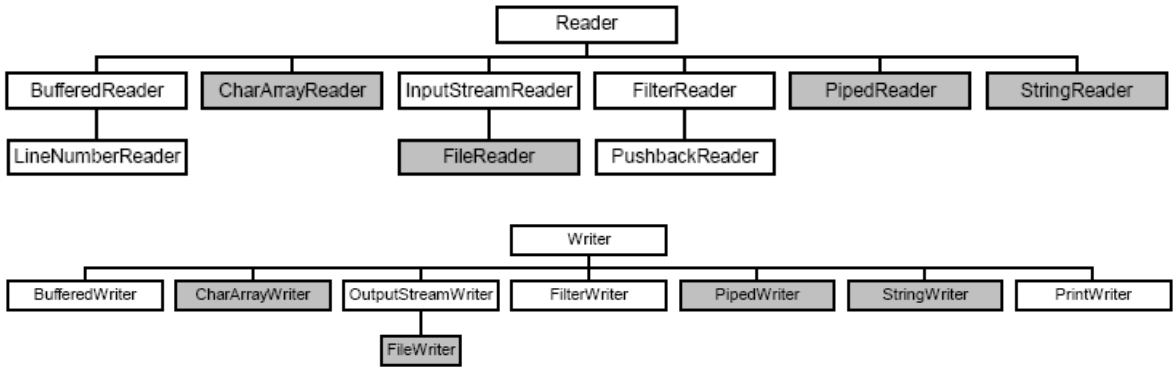


.1. Jerarquía de clases InputStream.



.2. Jerarquía de clases OutputStream.

En ***Java 1.1*** aparecieron dos nuevas familias de clases, derivadas de ***Reader*** y ***Writer***, que manejan ***caracteres*** en vez de ***bytes***. Estas clases resultan más prácticas para las aplicaciones en las que se maneja texto. Las clases que heredan de ***Reader*** y de ***Writer*** están incluidas en las siguientes figuras.



En las cuatro últimas figuras las clases con ***fondo gris*** definen de dónde o a dónde se están enviando los datos, es decir, el dispositivo con que conecta el ***stream***. Las demás (***fondo blanco***) añaden características particulares a la forma de enviarlos. La intención es que se combinen para obtener el comportamiento deseado. Por ejemplo:

```
BufferedReader in = new BufferedReader(new FileReader("autoexec.bat"));
```

Con esta línea se ha creado un ***stream*** que permite leer del archivo ***autoexec.bat***. Además, se ha creado a partir de él un objeto ***BufferedReader*** (que aporta la característica de utilizar ***buffer.-*** Un ***buffer*** es un espacio de memoria intermedia que actúa de "colchón" de datos. Cuando se necesita un dato del disco se trae a memoria ese dato y sus datos contiguos, de modo que la siguiente vez que se necesite algo del disco la probabilidad de que esté ya en memoria sea muy alta. Algo similar se hace para escritura, intentando realizar en una sola operación de escritura física varias sentencias individuales de escritura.-

Los caracteres que lleguen a través del ***FileReader*** pasarán a través del ***BufferedReader***, es decir utilizarán el ***buffer***.
A la hora de definir una comunicación con un dispositivo siempre se comenzará determinando el origen o destino de la comunicación (***clases en gris***) y luego se le añadirán otras características (***clases en blanco***).
Se recomienda utilizar siempre que sea posible las clases ***Reader*** y ***Writer***, dejando las de ***Java 1.0*** para cuando sean imprescindibles. Algunas tareas como la ***serialización*** y la ***compresión*** necesitan las clases ***InputStream*** y ***OutputStream***.

1.1 Los nombres de las clases de java.io

Las clases de ***java.io*** siguen una nomenclatura sistemática que permite deducir su función a partir de las palabras que componen el nombre, tal como se describe en la Tabla 1.

Palabra	Significado
InputStream, OutputStream	Lectura/Escritura de bytes
Reader, Writer	Lectura/Escritura de caracteres
File	Archivos
String, CharArray, ByteArray, StringBuffer	Memoria (a través del tipo primitivo indicado)
Piped	Tubo de datos
Buffered	Buffer
Filter	Filtro
Data	Intercambio de datos en formato propio de Java
Object	Persistencia de objetos
Print	Imprimir

1.2 Clases que indican el origen o destino de los datos

La Tabla 9.2 explica el uso de las clases que definen el lugar con que conecta el ***stream***.

Clases	Función que realizan
FileReader, FileWriter, FileInputStream y FileOutputStream	Son las clases que leen y escriben en **archivos** de disco. Se explicarán luego con más detalle.
StringReader, StringWriter, CharArrayReader, CharArrayWriter, ByteArrayInputStream, ByteArrayOutputStream, StringBufferInputStream	Estas clases tienen en común que se comunican con la **memoria** del ordenador. En vez de acceder del modo habitual al contenido de un String, por ejemplo, lo leen como si llegara carácter a carácter. Son útiles cuando se busca un modo general e idéntico de tratar con todos los dispositivos que maneja un programa.
PipedReader, PipedWriter, PipedInputStream, PipedOutputStream	Se utilizan como un "tubo" o conexión bilateral para transmisión de datos. Por ejemplo, en un programa con dos threads pueden permitir la comunicación entre ellos. Un thread tiene el objeto PipedReader y el otro el PipedWriter. Si los streams están conectados, lo que se escriba en el PipedWriter queda disponible para que se lea del PipedReader. Tambíen puede comunicar a dos programas distintos.

1.3 Clases que añaden características

La Tabla 9.3 explica las funciones de las clases que alteran el comportamiento de un ***stream*** ya definido.

Clases	Función que realizan
BufferedReader, BufferedWriter, BufferedInputStream, BufferedOutputStream	Como ya se ha dicho, añaden un **buffer** al manejo de los datos. Es decir, se reducen las operaciones directas sobre el dispositivo (lecturas de disco, comunicaciones por red), para hacer más eficiente su uso. ***BufferedReader*** por ejemplo tiene el método ***readLine()*** que lee una línea y la devuelve como un String.
InputStreamReader, OutputStreamWriter	Son clases puente que permiten **convertir** streams que utilizan bytes en otros que manejan caracteres. Son la única relación entre ambas jerarquías y no existen clases que realicen la transformación inversa.
ObjectInputStream, ObjectOutputStream	Pertenecen al mecanismo de la **serialización** y se explicarán más adelante.
FilterReader, FilterWriter, FilterInputStream, FilterOutputStream	Son clases base para aplicar diversos **filtros** o procesos al stream de datos. También se podrían extender para conseguir comportamientos a medida.
DataInputStream, DataOutputStream	Se utilizan para escribir y leer datos directamente en los formatos propios de Java. Los convierten en algo ilegible , pero independiente de plataforma y se usan por tanto para **almacenaje** o para **transmisiones** entre ordenadores de distinto funcionamiento.
PrintWriter, PrintStream	Tienen métodos adaptados para **imprimir** las variables de Java con la apariencia normal. A partir de un boolean escriben "true" o "false", colocan la coma de un número decimal, etc.

2 ENTRADA Y SALIDA ESTÁNDAR (TECLADO Y PANTALLA)

En ***Java***, la entrada desde teclado y la salida a pantalla están reguladas a través de la clase ***System***.

Esta clase pertenece al package ***java.lang*** y agrupa diversos métodos y objetos que tienen relación con el sistema local. Contiene, entre otros, tres objetos ***static*** que son:

System.in: Objeto de la clase ***InputStream*** preparado para recibir datos desde la entrada estándar del sistema (habitualmente el teclado).

System.out: Objeto de la clase ***PrintStream*** que imprimirá los datos en la salida estándar del sistema (normalmente asociado con la pantalla).

System.err: Objeto de la clase ***PrintStream***. Utilizado para mensajes de error que salen también por pantalla por defecto.

Estas clases permiten la comunicación alfanumérica con el programa a través de lo métodos incluidos en la Tabla 9.4. Son métodos que permiten la entrada/salida a un nivel muy elemental.

Métodos de System.in	Función que realizan
int read()	Lee un carácter y lo devuelve como int.
Métodos de System.out y System.err	**Función que realizan**
int print(cualquier tipo)	Imprime en pantalla el argumento que se le pase. Puede recibir cualquier tipo primitivo de variable de Java.
int println(cualquier tipo)	Como el anterior, pero añadiendo '\n' (nueva línea) al final.

Existen tres métodos de ***System*** que permiten sustituir la entrada y salida estándar. Por ejemplo, se utiliza para hacer que el programa lea de un archivo y no del teclado.

```
System.setIn(InputStream is);
System.setOut(PrintStream ps);
System.setErr(PrintStream ps);
```

El argumento de ***setIn()*** no tiene que ser necesariamente del tipo ***InputStream***. Es una referencia a la clase base, y por tanto puede apuntar a objetos de cualquiera de sus clases

derivadas (como ***FileInputStream***). Asimismo, el constructor de ***PrintStream*** acepta un ***OutputStream***, luego se puede dirigir la salida estándar a cualquiera de las clases definidas para salida.
Si se utilizan estas sentencias con un compilador de ***Java 1.1*** se obtiene un mensaje de método obsoleto (***deprecated***) al crear un objeto ***PrintStream***. Al señalar como obsoleto el constructor de esta clase se pretendía animar al uso de ***PrintWriter***, pero existen casos en los cuales es imprescindible un elemento ***PrintStream***. Afortunadamente, ***Java 1.2*** ha reconsiderado esta decisión y de nuevo se puede utilizar sin problemas.

2.1 Lectura desde teclado

Para leer desde teclado se puede utilizar el método ***System.in.read()*** de la clase ***InputStream***. Este método lee un carácter por cada llamada. Su valor de retorno es un ***int***. Si se espera cualquier otro tipo hay que hacer una conversión explícita mediante un ***cast***.

```
char c;
c=(char)System.in.read();
```

Este método puede lanzar la excepción ***java.io.IOException*** y siempre habrá que ocuparse de ella, por ejemplo en la forma:

```
try {
c=(char)System.in.read();
}
catch(java.io.IOException ioex) {
// qué hacer cuando ocurra la excepción
}
```

Para leer datos más largos que un simple carácter es necesario emplear un bucle ***while*** o ***for*** y unir los caracteres. Por ejemplo, para leer una línea completa se podría utilizar un bucle ***while*** guardando los caracteres leídos en un ***String*** o en un ***StringBuffer*** (más rápido que ***String***):

```
char c;
String frase = new String(""); // StringBuffer frase=new StringBuffer("");
try {
while((c=System.in.read()) != '\n')
frase = frase + c; // frase.append(c);
}
catch(java.io.IOException ioex) {}
```

Una vez que se lee una línea, ésta puede contener números de coma flotante, etc. Sin embargo, hay una manera más fácil de conseguir lo mismo: utilizar adecuadamente la librería ***java.io***.

3. LECTURA Y ESCRITURA DE ARCHIVOS

Aunque el manejo de archivos tiene características especiales, se puede utilizar lo dicho hasta ahora para las entradas y salidas estándar con pequeñas variaciones. ***Java*** ofrece las siguientes posibilidades:
Existen las clases ***FileInputStream*** y ***FileOutputStream*** (extendiendo ***InputStream*** y ***OutputStream***) que permiten leer y escribir ***bytes*** en archivos. Para archivos de texto son preferibles ***FileReader*** (desciende de ***Reader***) y ***FileWriter*** (desciende de ***Writer***), que realizan las mismas funciones. Se puede construir un objeto de cualquiera de estas cuatro clases a partir de un ***String*** que contenga el nombre o la dirección en disco del archivo o con un objeto de la clase ***File*** que representa dicho archivo. Por ejemplo el código

```
FileReader fr1 = new FileReader("archivo.txt");
```

es equivalente a:

```
File f = new File("archivo.txt");
```

```
FileReader fr2 = new FileReader(f);
```
Si no encuentran el archivo indicado, los constructores de ***FileReader*** y ***FileInputStream*** pueden lanzar la excepción ***java.io.FileNotFoundException***.
Los constructores de ***FileWriter*** y ***FileOutputStream*** pueden lanzar ***java.io.IOException***.
Si no encuentran el archivo indicado, lo crean nuevo. Por defecto, estas dos clases comienzan a escribir al comienzo del archivo. Para escribir detrás de lo que ya existe en el archivo ("append"), se utiliza un segundo argumento de tipo ***boolean*** con valor ***true***:
```
FileWriter fw = new FileWriter("archivo.txt", true);
```
Las clases que se explican a continuación permiten un manejo más fácil y eficiente que las vistas hasta ahora.

3.1 Lectura de archivos de texto

Se puede crear un objeto ***BufferedReader*** para leer de un archivo de texto de la siguiente manera:
```
BufferedReader br = new BufferedReader(new FileReader("archivo.txt"));
```
Utilizando el objeto de tipo ***BufferedReader*** se puede conseguir exactamente lo mismo que en las secciones anteriores utilizando el método ***readLine()*** y la clase ***StringTokenizer***. En el caso de archivos es muy importante utilizar el ***buffer*** puesto que la tarea de escribir en disco es muy lenta respecto a los procesos del programa y realizar las operaciones de lectura de golpe y no de una en una hace mucho más eficiente el acceso. Por ejemplo:
```
// Lee un archivo entero de la misma manera que de teclado
String texto = new String();
try {
FileReader fr = new FileReader("archivo.txt");
entrada = new BufferedReader(fr);
String s;
while((s = entrada.readLine()) != null)
texto += s;
entrada.close();
}
catch(java.io.FileNotFoundException fnfex) {
System.out.println("Archivo no encontrado: " + fnfex);}
catch(java.io.IOException ioex) {}
```

3.2 Escritura de archivos de texto

La clase ***PrintWriter*** es la más práctica para escribir un archivo de texto porque posee los métodos ***print***(cualquier tipo) y ***println***(cualquier tipo), idénticos a los de ***System.out*** (de clase ***PrintStream***).
Un objeto ***PrintWriter*** se puede crear a partir de un ***BufferedWriter*** (para disponer de ***buffer***), que se crea a partir del ***FileWriter*** al que se la pasa el nombre del archivo.
Después, escribir en el archivo es tan fácil como en pantalla. El siguiente ejemplo ilustra lo anterior:
```
try {
FileWriter fw = new FileWriter("escribeme.txt");
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter salida = new PrintWriter(bw);
salida.println("Hola, soy la primera línea");
salida.close();
// Modo append
bw = new BufferedWriter(new FileWriter("escribeme.txt", true));
salida = new PrintWriter(bw);
salida.print("Y yo soy la segunda. ");
double b = 123.45;
salida.println(b);
```

```
salida.close();
}
cacth(java.io.IOException ioex) { }
```

3.3 Archivos que no son de texto

DataInputStream y ***DataOutputStream*** son clases de ***Java 1.0*** que no han sido alteradas hasta ahora. Para leer y escribir datos primitivos directamente (sin convertir a/de ***String***) siguen siendo más útiles estas dos clases.
Son clases diseñadas para trabajar de manera conjunta. Una puede leer lo que la otra escribe, que en sí no es algo legible, sino el dato como una secuencia de ***bytes***. Por ello se utilizan para almacenar datos de manera independiente de la plataforma (o para mandarlos por una red entre ordenadores muy distintos).
El problema es que obligan a utilizar clases que descienden de ***InputStream*** y ***OutputStream*** y por lo tanto algo más complicadas de utilizar. El siguiente código primero escribe en el fichero *prueba.dat* para después leer los datos escritos:

```
// Escritura de una variable double
DataOutputStream dos = new DataOutputStream(
new BufferedOutputStream(
new FileOutputStream("prueba.dat")));
double d1 = 17/7;
dos.writeDouble(d);
dos.close();
// Lectura de la variable double
DataInputStream dis = new DataInputStream(
new BufferedInputStream(
new FileInputStream("prueba.dat")));
double d2 = dis.readDouble();
```

4. SERIALIZACIÓN

La ***serialización*** es un proceso por el que un objeto cualquiera se puede convertir en una ***secuencia de bytes*** con la que más tarde se podrá reconstruir dicho objeto manteniendo el valor de sus variables. Esto permite guardar un objeto en un archivo o mandarlo por la red.
Para que una clase pueda utilizar la serialización, debe implementar la interface ***Serializable***, que no define ningún método. Casi todas las clases estándar de ***Java*** son serializables. La clase MiClase se podría serializar declarándola como:

```
public class MiClase implements Serializable { }
```

Para escribir y leer objetos se utilizan las clases ***ObjectInputStream*** y ***ObjectOutputStream***, que cuentan con los métodos ***writeObject()*** y ***readObject()***. Por ejemplo:

```
ObjectOutputStream objout = new ObjectOutputStream(
new FileOutputStream("archivo.x"));
String s = new String("Me van a serializar");
objout.writeObject(s);
ObjectInputStream objin = new ObjectInputStream(new
FileInputStream("archivo.x"));
String s2 = (String)objin.readObject();
```

Es importante tener en cuenta que ***readObject()*** devuelve un ***Object*** sobre el que se deberá hacer un ***casting*** para que el objeto sea útil. La reconstrucción necesita que el archivo ****.class*** esté al alcance del programa (como mínimo para hacer este ***casting***).
Al serializar un objeto, automáticamente se serializan todas sus variables y objetos miembro.

A su vez se serializan los que estos objetos miembro puedan tener (todos deben ser serializables).También se reconstruyen de igual manera. Si se serializa un ***Vector*** que contiene varios ***Strings***, todo ello se convierte en una serie de ***bytes***. Al recuperarlo la reconstrucción deja todo en el lugar en que se guardó.
Si dos objetos contienen una referencia a otro, éste no se duplica si se escriben o leen ambos del mismo ***stream***. Es decir, si el mismo ***String*** estuviera contenido dos veces en el ***Vector***, sólo se guardaría una vez y al recuperarlo sólo se crearía un objeto con dos referencias contenidas en el vector.

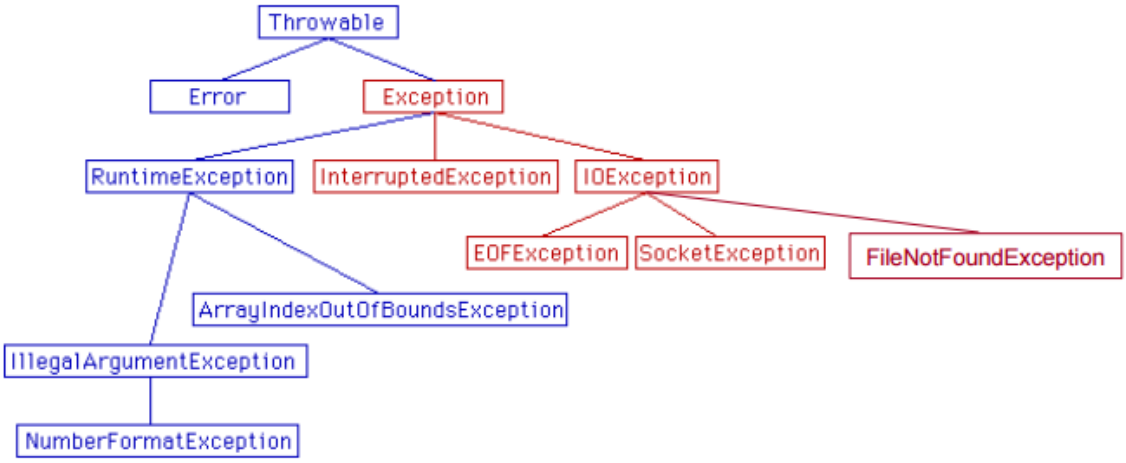
Control de la serialización
Aunque lo mejor de la serialización es que su comportamiento automático es bueno y sencillo, existe la posibilidad de especificar cómo se deben hacer las cosas.
La palabra clave ***transient*** permite indicar que un objeto o varible miembro no sea serializado con el resto del objeto. Al recuperarlo, lo que esté marcado como ***transient*** será ***0***, ***null*** o ***false*** (en esta operación no se llama a ningún constructor) hasta que se le dé un nuevo valor. Podría ser el caso de un ***password*** que no se quiere guardar por seguridad.
Las variables y objetos ***static*** no son serializados. Si se quieren incluir hay que escribir el código que lo haga. Por ejemplo, habrá que programar un método que serialice los objetos estáticos al que se llamará después de serializar el resto de los elementos. También habría que recuperarlos explícitamente después de recuperar el resto de los objetos.
Las clases que implementan ***Serializable*** pueden definir dos métodos con los que controlar la serialización. No están obligadas a hacerlo porque una clase sin estos métodos obtiene directamente el comportamiento por defecto. Si los define serán los que se utilicen al serializar:

```
private void writeObject(ObjectOutputStream stream) throws IOException
private void readObject(ObjectInputStream stream) throws IOException
```

El primero permite indicar qué se escribe o añadir otras instrucciones al comportamiento por defecto. El segundo debe poder leer lo que escribe ***writeObject()***. Puede usarse por ejemplo para poner al día las variables que lo necesiten al ser recuperado un objeto. Hay que leer en el mismo orden en que se escribieron los objetos.
Se puede obtener el comportamiento por defecto dentro de estos métodos llamando a ***stream.defaultWriteObject()*** y ***stream.defaultReadObject()***.
Para guardar explícitamente los tipos primitivos se puede utilizar los métodos que proporcionan ***ObjectInputStream*** y ***ObjectOutputStream***, idénticos a los de ***DataInputStream*** y ***DataOutputStream*** (***writeInt()***, ***readDouble()***, ...) o guardar objetos de sus clases equivalentes (***Integer***, ***Double***...).
Por ejemplo, si en una clase llamada ***Tierra*** se necesita que al serializar un objeto siempre le acompañe la constante ***g*** (9,8) definida como ***static*** el código podría ser:

```
static double g = 9.8;
private void writeObject(ObjectOutputStream stream) throws IOException {
stream.defaultWriteObject();
stream.writeDouble(g);
}
private void readObject(ObjectInputStream stream) throws IOException {
stream.defaultReadObject();
g = stream.readDouble(g);
}
```

5. Control de Excepciones de Entrada/Salida



En rojo: son las excepciones que es obligatorio controlar. Checked exceptions
En azul: son las excepciones que no es obligatorio controlar. Unchecked exceptions