Programación en Lenguaje Java

Tema 10. Entrada/Salida con ficheros

Michael González Harbour

Mario Aldea Rivas

Departamento de Matemáticas,
Estadística y Computación

Programación en Java

1 ..., 2 ...

3. Estructuras algorítmicas

4. Datos Compuestos

5. Entrada/salida

6. Clases, referencias y objetos

7. Modularidad y abstracción

8. Herencia y polimorfismo

9. Tratamiento de errores

10. Entrada/salida con ficheros

- Ficheros. Flujos de datos (streams). Escritura de ficheros de texto. Lectura de ficheros de texto. Escritura de ficheros binarios. Lectura de ficheros binarios. Ficheros binarios de objetos. Resumen de tipos de ficheros

11. Pruebas

10.1 Ficheros

Fichero:

- secuencia de bytes en un dispositivo de almacenamiento: disco duro, memoria USB, CD, DVD, ...
- se puede leer y/o escribir
- se identifica mediante un nombre (*pathname*)
  - `/home/pepe/documentos/un_fichero`

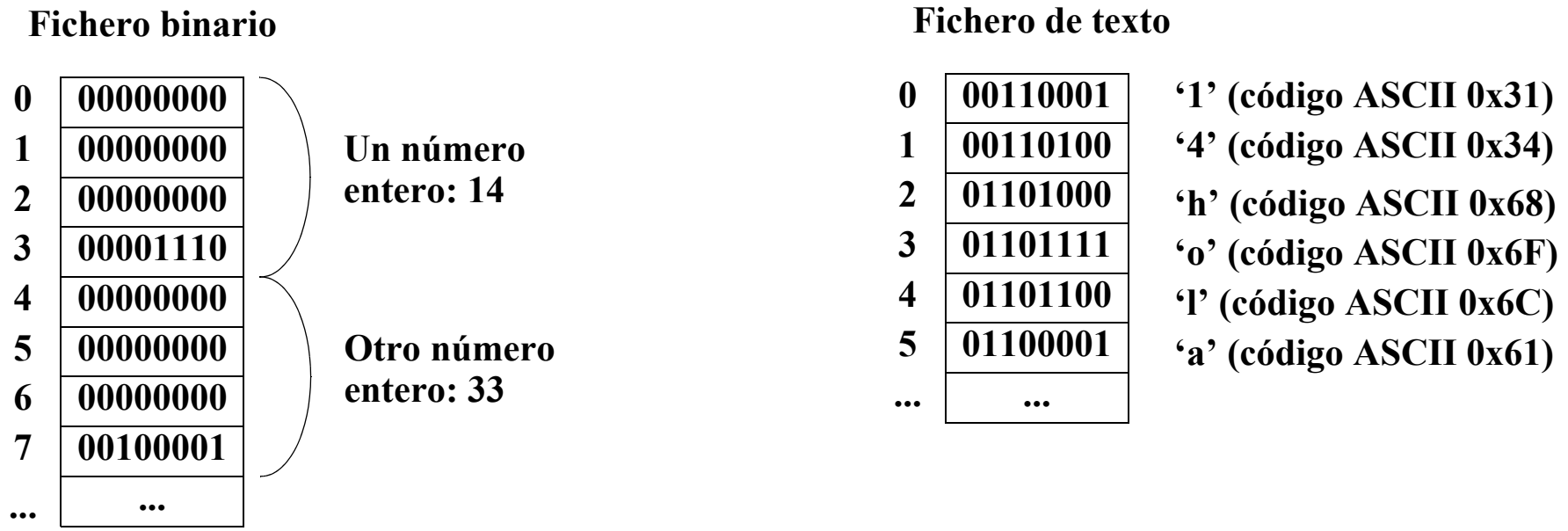
Tipos de ficheros:

- programas: contienen instrucciones
- datos: contienen información, como números (enteros o reales), secuencias de caracteres, ...
- en algunos sistemas operativos (como Linux) también son ficheros los directorios, los dispositivos, las tuberías, ...

Ficheros de texto y binarios

Tipos de ficheros de datos:

- ***de bytes*** (binarios): pensados para ser leídos por un programa
- ***de caracteres*** (de texto): pueden ser leídos y escritos por una persona



- Para "entender" los contenidos de un fichero es necesario conocer de antemano el tipo de datos que contiene

Punteros de lectura y escritura

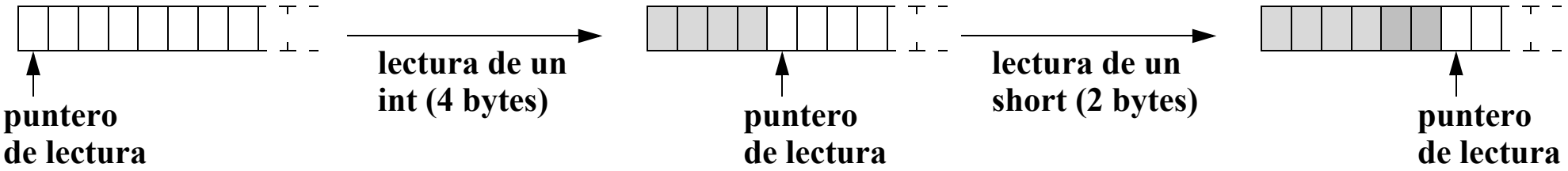
Indican el próximo byte a leer o a escribir

Gestionados automáticamente por el sistema operativo

Comienzan apuntando al primer byte del fichero

Van avanzando por el fichero según se van leyendo/escribiendo sus contenidos

Ejemplo:



10.2 Flujos de datos (*streams*)

La Entrada/Salida de Java se organiza generalmente mediante objetos llamados `Streams`

Un `Stream` es la generalización de un fichero:

- secuencia ordenada de datos con un determinado origen y destino
- su origen o destino puede ser un fichero, pero también un string o un dispositivo (p.e. el teclado)

Para poder usar un *stream* primero hay que ***abrirle***

- se abre en el momento de su creación
- y hay que ***cerrarle*** cuando se deja de utilizar

Las clases relacionadas con *streams* se encuentran definidas en el paquete `java.io` (`io` es la abreviatura de *Input/Output*)

Clasificación de los *streams*

Por el tipo de datos que "transportan":

- ***binarios*** (de bytes)
- ***de caracteres*** (de texto)

Por el sentido del flujo de datos:

- ***de entrada***: los datos fluyen desde el dispositivo o fichero hacia el programa
- ***de salida***: los datos fluyen desde el programa al dispositivo

Según su cercanía al dispositivo:

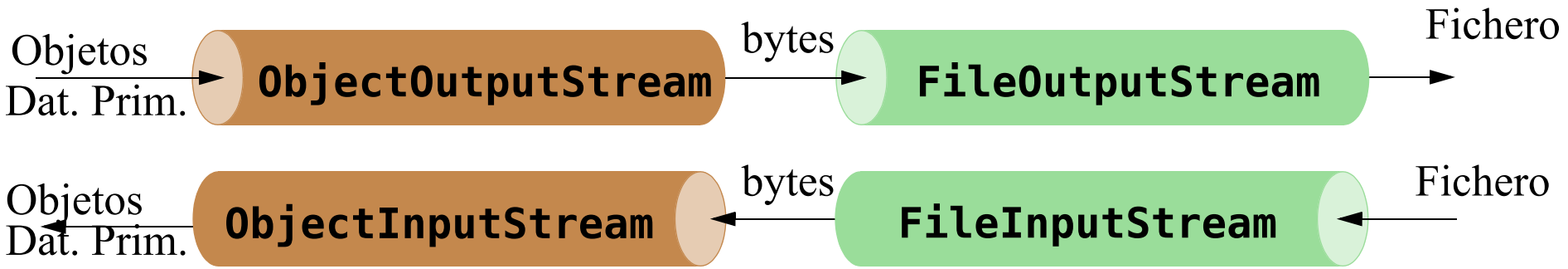
- ***iniciadores***: son los que directamente vuelcan o recogen los datos del dispositivo
- ***filtros***: se sitúan entre un *stream* iniciador y el programa

Uso de los Streams

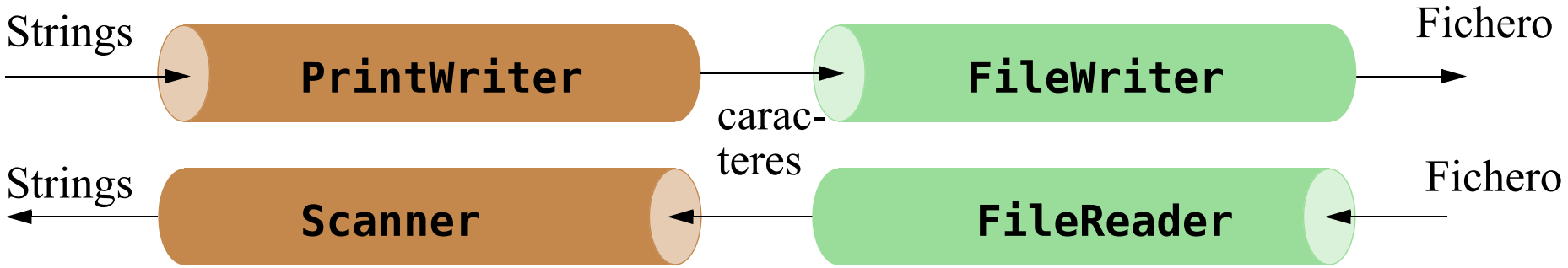
Normalmente se utilizan por parejas

- formadas por un *stream* iniciador y un filtro

Binarios



De Texto:



Objetos *stream* predefinidos

`System.out`: **Salida estándar (consola)**

- objeto de la clase `PrintStream` (subclase de `OutputStream`)
  - métodos `print`, `println`, `printf`, ...

`System.err`: **Salida de error (consola)**

- también es un objeto de la clase `PrintStream`

`System.in`: Entrada estándar (teclado)

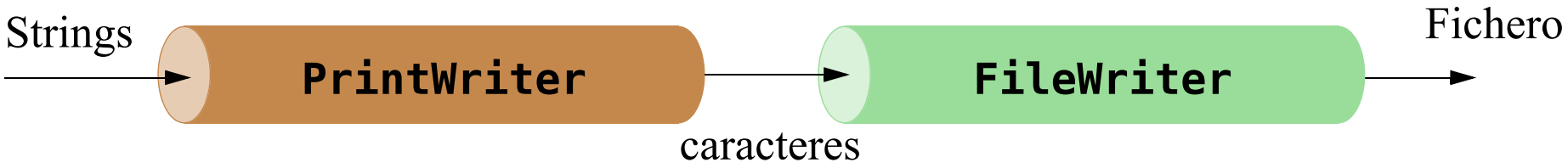
- objeto de la clase `InputStream`

Deberían ser de las clases `PrintWriter` y `BufferedReader`

- pero los *streams* de caracteres no existían en las primeras versiones de Java
- siguen siendo *streams* binarios por compatibilidad con versiones antiguas

10.3 Escritura de ficheros de texto

Pareja de *streams*: `PrintWriter` (filtro) y `FileWriter` (iniciador)



Esquema general de uso:

```
PrintWriter out = null;
try {
  // Abre el fichero (crea los streams y los conecta)
  out = new PrintWriter(new FileWriter(nomFich));
  // escribe en el fichero
  ... diferente en cada caso ...
} finally {
  if (out != null)
    out.close(); // cierra el fichero (cierra el stream)
}
```

Clase `FileWriter`

Operaciones más habituales:

Descripción	Declaración
Constructor. Requiere el nombre del fichero. Lo crea si no existe. Si existe se borran sus contenidos. Lanza `IOException` si el fichero no se puede crear	`FileWriter(String s)` `  throws IOException`

Igual que el anterior, salvo en que cuando `añade` es `true` no se borran los contenidos, sino que los datos se añaden al final del fichero	`FileWriter(String s,`
	`  boolean añade)`
	`  throws IOException`

Clase `PrintWriter`

Operaciones más habituales:

Descripción	Declaración
Constructor. Requiere un `Writer`	`PrintWriter(Writer writer)`
Escribir un string	`void print(String str)`
Escribir un string con retorno de línea	`void println(String str)`
Escribe los argumentos con el formato deseado	`printf(String formato, Object... args)`
Sincroniza e informa si ha habido un error	`boolean checkError()`
Sincronizar	`void flush()`
Cerrar	`void close()`

- Los métodos ***no lanzan*** `IOException`:
  - para saber si ha habido un error hay que llamar a `checkError`

Ejemplo: escritura fichero de texto

```
static void ejemploEscribeFichTexto(String nomFich,
    int i, double x, String str) throws IOException {
  PrintWriter out = null;
  try {
    // Abre el fichero
    out = new PrintWriter(new FileWriter(nomFich));
    // escribe los datos en el fichero
    out.println("Entero: " + i + "   Real: " + x);
    out.println("String: " + str);
  } finally {
    if (out != null)
      out.close(); // Cierra el fichero
  }
}
```

Fichero generado:

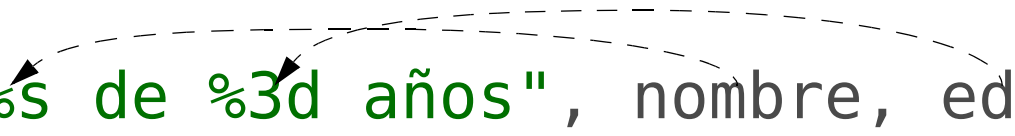
```
Entero: 11   Real: 22.2
String: hola
```

Escritura de texto con formato

La clase `PrintWriter` dispone de una operación de salida de texto con formato, llamada `printf`

- el objeto `System.out` que representa la pantalla, también
- está copiada del lenguaje C
- el primer parámetro es el ***string de formato***
- los siguientes son un número variable de parámetros

Ejemplo



```
System.out.printf("%s de %3d años", nombre, edad);
```

Produce la salida (suponiendo nombre="Pedro", edad=18):

```
Pedro de  18 años
```

String de formato

Contiene caracteres que se muestran tal cual

- y ***especificaciones de formato*** que se sustituyen por los sucesivos parámetros

Especificaciones de formato más habituales:

```
%d   enteros
%c   caracteres
%s   string
%f   float y double, coma fija
%e   float y double, notación exponencial
%g   float y double, exponencial o coma fija
%n   salto de línea en el formato del sist. operat.
%%   el carácter %
```

Puede lanzarse `IllegalFormatException` si el formato no corresponde al parámetro

Después del carácter **%** se puede poner un carácter de opciones:

```
-       alinear a la izquierda
0       rellenar con ceros (números sólo)
+       poner signo siempre (números sólo)
```

Para forzar la utilización del punto como separador de las cifras decimales:

```
import java.util.Locale;
...
  Locale.setDefault(Locale.ENGLISH);
  ... // usa printf
```

Especificación de anchura y precisión

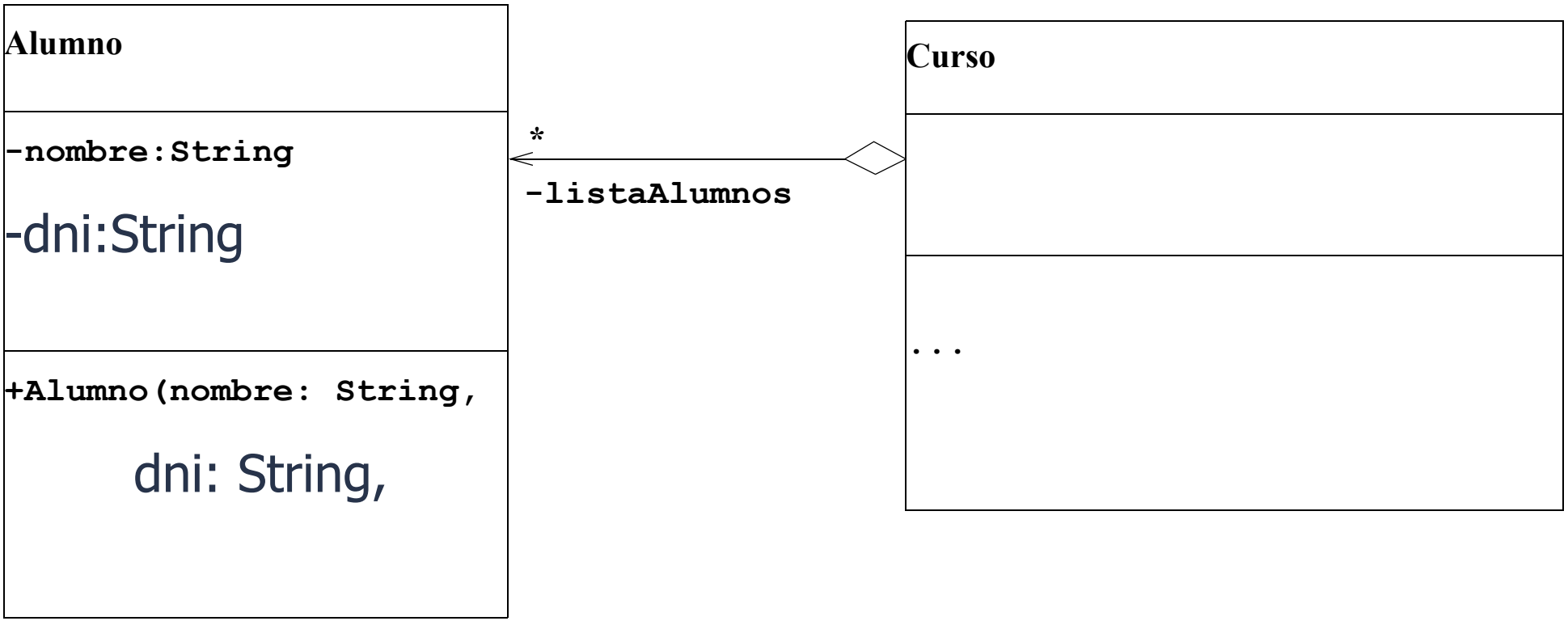
Puede añadirse después del "*%*" (y el carácter de opción si lo hay) la especificación de anchura mínima y/o número de decimales; ejemplos

Invocación de printf()	Salida
`printf("Pi= %4.0f %n", Math.PI);`	`Pi=    3`
`printf("Pi= %4.2f %n", Math.PI);`	`Pi= 3.14`
`printf("Pi= %12.4f %n", Math.PI);`	`Pi=       3.1416`
`printf("Pi= %12.8f %n", Math.PI);`	`Pi=   3.14159265`
`printf("I= %8d %n", 18);`	`I=       18`
`printf("I= %4d %n", 18);`	`I=   18`
`printf("I= %04d %n", 18);`	`I= 0018`

Ejemplo: escritura de ficheros de texto con formato (método `printf`)

Añadir el método `generaListado` a la clase `Curso`:

- Escribe en un fichero de texto los datos de todos los alumnos del curso alineando en columnas el nombre, el DNI y la nota.



Ejemplo: escritura de ficheros de texto con formato (método `printf`) (cont.)

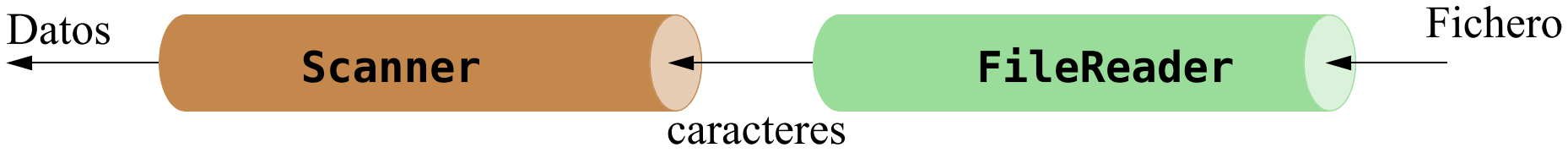
```
public void generaListado(String nomFich)
                                        throws IOException {
  PrintWriter out = null;
  try {
    // abre el fichero de texto
    out = new PrintWriter(new FileWriter(nomFich));
    // escribe el listado alumno por alumno
    for(Alumno a: listaAlumnos) {
      // nombre con 25 carac. justificado a la izq.
      // nota con 4 carac. totales con un decimal
      out.printf("%-25s  DNI:%s  Nota:%4.1f%n",
                 a.nombre(),a.dni(), a.nota());
    }
  } finally {
    if (out!=null)
      out.close();
  }
}
```

Fichero de texto generado:

```
Pepe García Hernández       DNI:123456789  Nota: 5.0
Lolo Hernández García       DNI:234567890  Nota: 0.0
Manu López Gómez            DNI:345678901  Nota:10.0
Pepito Gómez López          DNI:456789012  Nota: 7.5
```

10.4 Lectura de ficheros de texto

La lectura de un fichero de texto se realiza con la pareja de *streams* `Scanner` (filtro) y `FileReader` (iniciador)



`Scanner` se comporta como un *stream* filtro de caracteres

- aunque realmente no lo es, ya que no extiende a `Reader`

Clase `FileReader`

Operaciones habituales:

Descripción	Declaración
Constructor. Requiere el nombre del fichero. Si no existe lanza `FileNotFoundException`	`FileReader(String s) throws FileNotFoundException`

Clase Scanner

La clase `Scanner` (paquete `java.util`) permite leer números y texto de un fichero de texto y de otras fuentes

- permite la lectura del texto línea a línea
- permite la lectura sencilla de números y palabras separadas por el separador especificado
  - el separador por defecto es cualquier tipo de espacio en blanco (espacio, salto de línea, tabulador, etc.)
  - puede utilizarse otro separador utilizando el método `useDelimiter`
- permite reconocer patrones de texto conocidos como "expresiones regulares" (no lo veremos en esta asignatura)

Principales operaciones de la clase Scanner

Descripción	Declaración
Constructor. Requiere un `InputStream`	`Scanner(InputStream source)`
Constructor. Requiere un objeto que implemente `Readable` (por ejemplo un `FileReader`)	`Scanner(Readable source)`
Constructor. Requiere un String	`Scanner(String source)`
Cerrar	`void close()`
Configura el formato de los números. Usar `Locale.ENGLISH` para leer números que utilicen el carácter '`.`' como separador decimal. Usar `Locale.FRENCH` para leer números que utilicen el carácter '`,`' como separador decimal.	`Scanner useLocale(` `Locale locale)`

Descripción	Declaración
Leer una línea	`String nextLine()`
Indica si quedan más líneas por leer	`boolean hasNextLine()`
Leer un booleano	`boolean nextBoolean()`
Indica si es posible leer una palabra que se interprete como un booleano	`boolean hasNextBoolean()`
Leer una palabra	`String next()`
Indica si quedan más palabras o datos por leer	`boolean hasNext()`
Leer un double	`double nextDouble()`
Indica si es posible leer una palabra que se interprete como un double	`boolean hasNextDouble()`
Leer un int	`int nextInt()`
Indica si es posible leer una palabra que se interprete como int	`boolean hasNextInt()`
Cambia el delimitador que separa los ítems	`Scanner useDelimiter(String pattern)`

Excepciones que pueden lanzar

- `NoSuchElementException`: no quedan más palabras
- `IllegalStateException`: el *Scanner* está cerrado
- `InputMismatchException`: el dato leído no es del tipo esperado

Ejemplo con la clase Scanner

- **Para el fichero:**

```
azul 1.0 3.5 7.7
rojo 2
verde 10.0 11.1
```

- **Se desea obtener la siguiente salida por consola:**

```
Palabra:azul
Número:1.0
Número:3.5
Número:7.7
Palabra:rojo
Número:2.0
Palabra:verde
Número:10.0
Número:11.1
```

```
private static void muestraContenidoFich(
  String nomFich) throws FileNotFoundException {

  Scanner in = null;

  try {
    // abre el fichero
    in = new Scanner(new FileReader(nomFich));

    // configura el formato de números
    in.useLocale(Locale.ENGLISH);

    // lee el fichero palabra a palabra
    while (in.hasNext()) {
      // lee primera palabra
      String palabra = in.next();

      System.out.println("Palabra:" + palabra);
```

```
        // lee los números después de la palabra
        while (in.hasNextDouble()) {
          // lee un double
          double d = in.nextDouble();

          System.out.println("Número:"+d);
        }
      } // while (in.hasNext())

    } finally {
      if (in != null){
        in.close();
      }
    } // try

}  // método
```

Procesado de Strings con la clase Scanner

La clase Scanner también puede ser utilizada para procesar Strings de una manera sencilla

Ejemplo: procesado de Strings con Scanner

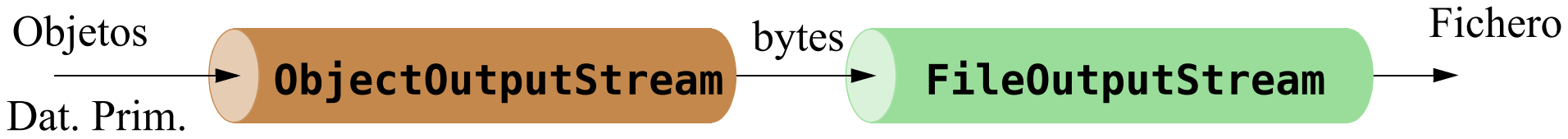
```
// pide datos al usuario
Lectura lect = new Lectura("Marcas personales:");
lect.creaEntrada("Nombre y marcas",
                 "Pepe  10.4 11.2  10.2");
lect.esperaYCierra("Introduce nombre y marcas");
String strDatos = lect.leeString("Nombre y marcas");
// utiliza la clase scanner para procesar el string
Scanner scn = new Scanner(strDatos);
try {
  // la primera palabra es el nombre
  String nombre = scn.next();
  // las siguientes son las marcas, las vamos
  // sumando para calcular la media
  int n = 0; // número de marcas
  double suma = 0;
```

```
    // comienza a leer las marcas, si no hay ninguna
    // se lanza NoSuchElementException
    do {
      n++;
      suma += scn.nextDouble();
    } while (scn.hasNext());

    // muestra el resultado final
    System.out.printf("Marca media de %s: %1.2f",
                      nombre, suma/n);
  } catch (InputMismatchException e) {
    System.out.println(
        "Error: una de las marcas no es un número");
  } catch (NoSuchElementException e) {
    System.out.println(
       "Error: debes introducir al menos una marca");
  }
```

10.5 Escritura de ficheros binarios

Se usa la pareja de *streams* `FileOutputStream` (iniciador) y `ObjectOutputStream` (filtro)



Esquema general de uso:

```
ObjectOutputStream out = null;
try {
  // crea los streams y los conecta
  out = new ObjectOutputStream(new FileOutputStream(nomFich));
  // escribe en el fichero
  ... diferente en cada caso ...
} finally {
  if (out != null)
    out.close();
}
```

Clase `FileOutputStream`

Operaciones más habituales:

Descripción	Declaración
Constructor. Requiere el nombre del fichero. Lo crea si no existe. Si existe se borran sus contenidos. Lanza `FileNotFoundException` si el fichero no se puede crear	`FileOutputStream(String s) throws FileNotFoundException`
Igual que el anterior, salvo en que cuando `añade` es `true` no se borran los contenidos, sino que los datos se añaden al final del fichero	`FileOutputStream(String s, boolean añade) throws FileNotFoundException`

Clase `ObjectOutputStream`

Operaciones más habituales:

Descripción	Declaración
Constructor. Requiere un `OutputStream`	`ObjectOutputStream(` `  OutputStream out)`
Escribir un booleano	`void writeBoolean(boolean b)`
Escribir un double	`void writeDouble(double d)`
Escribir un int	`void writeInt(int i)`
Escribir un objeto (incluido strings) Se escriben también los objetos a los que el objeto `obj` se refiere (y así recursivamente)	`void writeObject(Object obj)`
Sincronizar (llama a `out.flush()`)	`void flush()`
Cerrar (llama a `out.close()`)	`void close()`

Todos los métodos (incluido el constructor) lanzan `IOException`

- error al acceder al `OutputStream` (normalmente un fichero)

Ejemplo: escritura de fichero binario de tipos primitivos

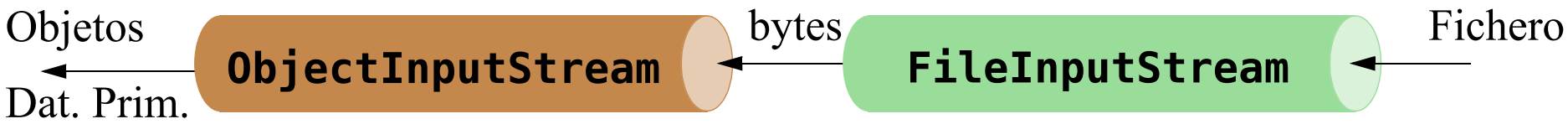
```
ObjectOutputStream sal = null;
try {
  // abre los streams iniciador y filtro
  sal = new ObjectOutputStream(
                new FileOutputStream("fich.dat"));

  // escribe varios datos
  sal.writeInt(65);
  sal.writeBoolean(true);
  sal.writeDouble(2.0);
} finally {
  if (sal != null) {
    sal.close(); // cierra los streams
  }
}
```

10.6 Lectura de ficheros binarios

Es posible leer variables y objetos de un fichero binario que fue creado según lo expuesto en el apartado anterior

Se usa la pareja de *streams* `FileInputStream` (iniciador) y `ObjectInputStream` (filtro)



Clase `FileInputStream`

Operaciones más habituales:

Descripción	Declaración
Constructor. Requiere el nombre del fichero. Si el fichero no existe lanza `FileNotFoundException`	`FileInputStream(String s)` `  throws FileNotFoundException`

Clase ObjectInputStream

Descripción	Declaración
Constructor. Requiere un `InputStream`	`ObjectInputStream (InputStream in)`
Leer un booleano	`boolean readBoolean()`
Leer un double	`double readDouble()`
Leer un int	`int readInt()`
Leer un objeto (incluido strings). Se leen también los objetos a los que el objeto se refiere (recursivamente)	`Object readObject()`
Número de bytes hasta el fin de fichero	`int available()`
Cerrar	`void close()`

- `IOException`: problema al acceder al `InputStream`
- `EOFException`: alcanzado el fin de fichero
- `ClassNotFoundException`: sólo producida por `readObject`

Ejemplo: lectura de fichero binario de tipos primitivos

```
int i; boolean b; double d;
ObjectInputStream ent = null;
try {
  // abre los streams iniciador y filtro
  ent = new ObjectInputStream(
                new FileInputStream("fich.dat"));
  // lee los datos
  i = ent.readInt();
  b = ent.readBoolean();
  d = ent.readDouble();
} finally {
  if (ent != null) {
    ent.close(); // cierra los streams
  }
}
```

Ejemplo: uso de `available()`

Lee todos los números enteros que hay en un fichero y les retorna en un `ArrayList`

```
private static ArrayList<Integer> leeNumeros(String nomFich)
                                  throws FileNotFoundException, IOException {
   // ArrayList a retornar
   ArrayList<Integer> numeros = new ArrayList<Integer>();
   ObjectInputStream ent = null;
   try {
      // abre el fichero binario
      ent = new ObjectInputStream(new FileInputStream(nomFich));
      // añade al ArrayList todos los números que contiene el fichero
      while(ent.available() > 0) {
         numeros.add(ent.readInt());
      }
   } finally {
      if (ent != null) {
         ent.close(); // cierra el fichero
      }
   }
   return numeros;
}
```

10.7 Ficheros binarios de objetos

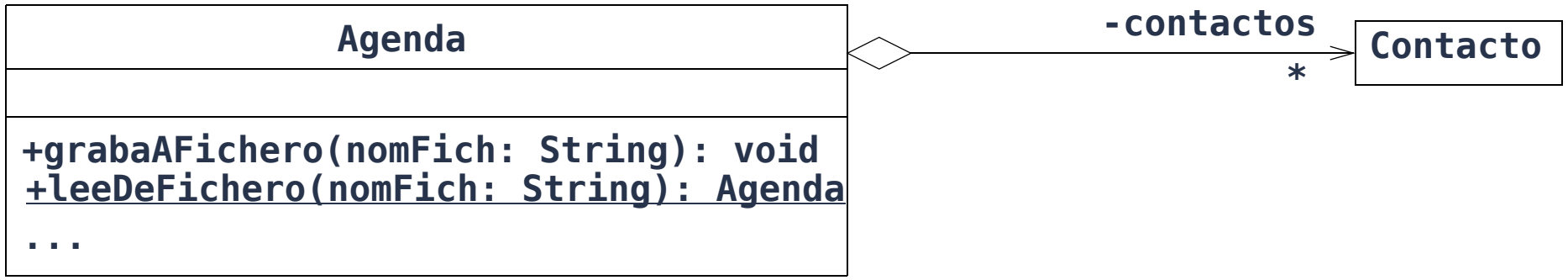
Un ***tipo especial de ficheros binarios*** proporcionados por Java

- forma muy sencilla de grabar/recuperar el estado de un programa
- con una sola instrucción se graba/recupera un objeto y, recursivamente, todos los objetos a los que éste hace referencia

Para poder escribir un objeto su clase debe implementar la interfaz `Serializable`, de la manera siguiente:

```
import java.io.*;
public class Persona implements Serializable
{...}
```

Ejemplo: agenda de contactos



Añadir a la clase Agenda operaciones para grabar/recuperar la agenda (incluyendo los contactos que contiene)

- Previamente ha sido necesario hacer "serializables" todas las clases que se van a grabar:

```
import java.io.*;
public class Contacto implements Serializable {...

import java.io.*;
public class Agenda implements Serializable {...
```

```
public void grabaAFichero(String nomFich)
                                    throws IOException {
   ObjectOutputStream sal = null;
   try {
      // abre los streams iniciador y filtro
      sal = new ObjectOutputStream(
                        new FileOutputStream(nomFich));
      // graba el objeto actual
      sal.writeObject(this);
   } finally {
      if (sal != null) {
         sal.close(); // cierra los streams
      }
   }
}
```

```
public static Agenda leeDeFichero(String nomFich)
       throws IOException, ClassNotFoundException {
  ObjectInputStream ent = null;
  try {
    // abre el fichero
    ent = new ObjectInputStream(
                          new FileInputStream(nomFich));
    // lee el objeto y le retorna
    return (Agenda)ent.readObject();
  } finally {
    if (ent != null) {
      ent.close(); // cierra los streams
    }
  }
}
```

10.8 Resumen de tipos de ficheros

Ficheros de ***texto***:

+ Pueden ser editados/leídos por una persona
+ Portables entre distintos lenguajes de programación, sistemas operativos y arquitecturas hardware
- Requieren más espacio que los binarios

Ficheros ***binarios***

+ Requieren menos espacio que los de texto
- No pueden ser editados/leídos por una persona
- Posibles problemas de portabilidad (formato de números en coma flotante, little-endian/big-endian, etc.)

Ficheros ***binarios de objetos***

+ Mecanismo sencillo y potente
- No portables (sólo para Java)