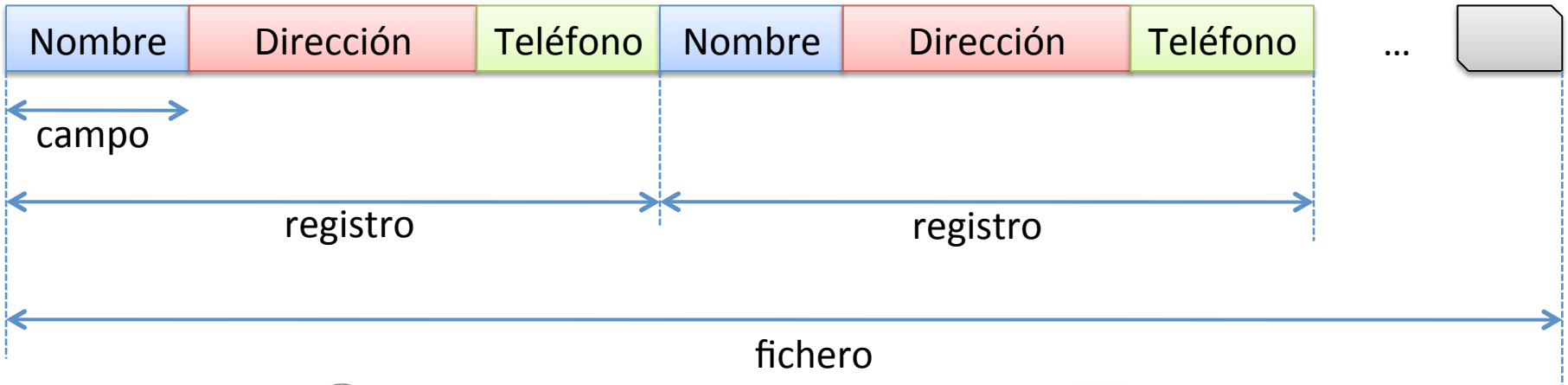
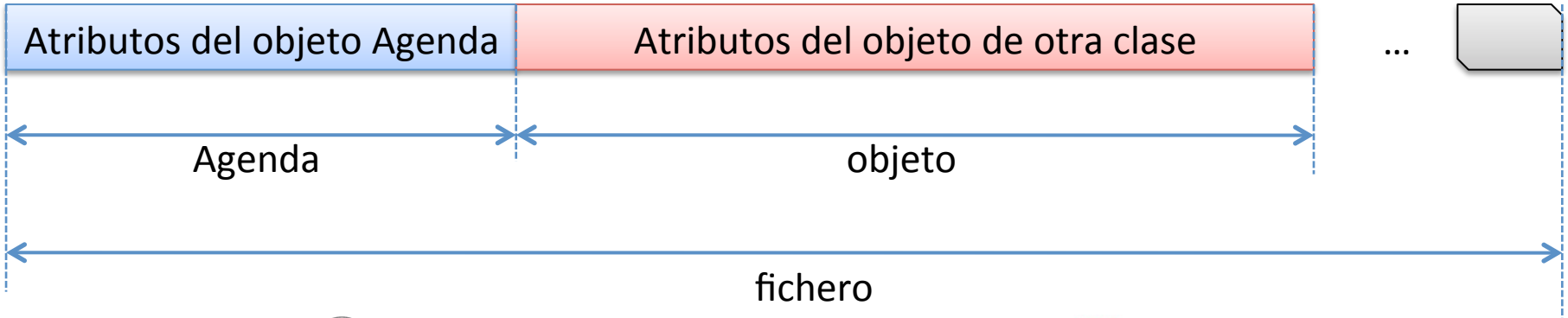
Ficheros Binarios de Acceso Secuencial

- Con ellos es posible almacenar y recuperar, de forma codificada, datos en ficheros (así, un int ocupará 4 bytes, un double 8, etc.).
- En general se debe seguir una política en el almacenamiento que facilite después la recuperación adecuada de la información.
- Por ejemplo: si se salva el contenido de una agenda, almacenando los valores elementales de cada uno de sus componentes, se puede seguir una política de agrupación de la información en los denominados registros (grupos de información similar que se repiten):



Ficheros Binarios de Acceso Secuencial

- En Java también es posible almacenar una secuencia de objetos para poder recuperarlos posteriormente. Esto se denomina **serialización**.
- Por ejemplo, es posible salvar el contenido de una agenda (objeto de la clase `Agenda`) almacenando uno a uno sus objetos constituyentes (objetos de tipo `ItemAgenda`).
- O incluso se podría salvar en el fichero un objeto de tipo `Agenda` (que incluiría internamente todos los `ItemAgenda` que lo forman.
- Además, un fichero puede contener objetos pertenecientes a distintas clases o incluso incluir objetos y valores elementales simultáneamente.



Ficheros Binarios de Acceso Secuencial

- En Java es posible escribir o leer en ellos en formato binario:
  - Valores de tipos primitivos como boolean, int, double, etc.,
  - Objetos, con todos sus componentes internos, incluyendo todos los objetos referenciados por los primeros (y así sucesivamente, de forma recursiva).
- Se utilizan para ello las clases java.io.ObjectInputStream (lectura) y java.io.ObjectOutputStream (escritura) del paquete java.io.

Atención: para poder escribir o leer los objetos de una clase en un stream es necesario que en la declaración de la misma se diga que **implementa** el interfaz **serializable**, lo que se consigue declarándolo en su cabecera. Por ejemplo:

```
public class ItemAgenda implements Serializable {
   ……
} // fin de la clase ItemAgenda
```

Ficheros Binarios de Acceso Secuencial

- Para la lectura o escritura de un fichero binario se debe:
  1. Crear un File con el origen / destino de datos.
  2. Envolverlo en un FileInputStream / FileOutputStream para crear un flujo de datos desde / hacia el fichero.
  3. Envolver el objeto anterior en un ObjectInputStream / ObjectOutputStream para poder leer / escribir tipos de datos primitivos u objetos del flujo de datos.

- Posteriormente, para escribir o leer se deben usar métodos como:
  - writeInt, writeDouble, writeBoolean, writeObject, etc. (en escritura)
  - readInt, readDouble, readBoolean, readObject, etc. (en lectura)

Escritura y lectura de Datos Binarios

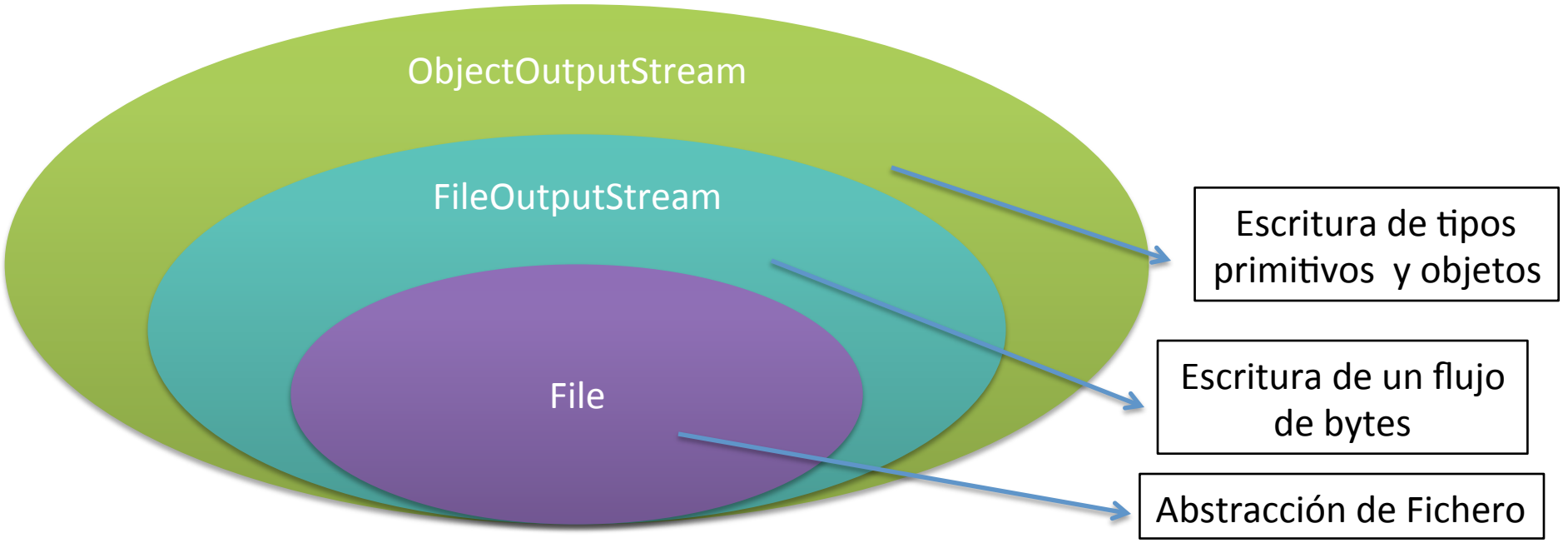
- La clase ObjectOutputStream contiene métodos para escribir tipos primitivos y objetos a un stream.
- Existen métodos análogos en ObjectInputStream para leerlos.

Método / Constructor	Descripción
ObjectOutputStream(OutputStream out)	Crea un nuevo objeto a partir del stream de salida
void **writeInt**(int v)	Escribe el entero v al stream de salida como 4 bytes
void **writeLong**(long v)	Escribe el long v al stream de salida como 8 bytes
void **writeUTF**(String str)	Escribe el String str en formato portable (UTF8 mod.)
void **writeDouble**(double v)	Escribe el double v al stream de salida como 8 bytes
void **writeObject**(Object obj)	Escribe el Object obj al stream de salida. Lo que provoca la escritura de todos los objetos de los que obj esté compuesto (y así recursivamente).

- Los métodos de escritura pueden lanzar la excepción IOException.

Los ficheros binarios creados con ObjectOutputStream sólo podrán ser leídos con un ObjectInputStream, debido a las conversiones de formato que se efectúan.

Uso de ObjectOutputStream



- Ejemplo de instanciación para escritura de datos primitivos u objetos a partir de un fichero binario (la lectura es análoga):
  - ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(new File(fichero)));
  - out.writeInt(45);

Es posible evitar la creación del File, pasando el nombre del fichero al constructor de FileOutputStream

Gestión de Excepciones

- El constructor de FileOutputStream puede lanzar la excepción *FileNotFoundException*.
  - Si el fichero especificado no existe en el sistema de archivos.
- Los métodos de escritura de ObjectOutputStream y los métodos de lectura de ObjectInputStream pueden lanzar alguna excepción de tipo *IOException* o de alguna subclase de la misma.
  - Si ocurre algún problema al escribir el fichero (permisos, hw, etc.).
  - Si la clase del objeto que se quiere escribir no es serializable.
- Además, el método readObject() de la clase ObjectInputStream puede lanzar una *ClassNotFoundException* si no se puede determinar la clase del objeto que se intenta leer.
- Todas estas excepciones deben ser gestionadas mediante el uso adecuado de un bloque try-catch.

Ejemplo de uso de Ficheros Binarios Secuenciales de Valores Elementales

- Programa que escribe datos en un fichero binario y posteriormente los lee antes de mostrarlos.

```
import java.io.*;
class Calificaciones {
  public static void main(String[] args){
   String fichero = "calificaciones.data";  String nombre = “PRG"; int conv = 1; double nota = 7.8;
   try {
      ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(fichero));
      out.writeUTF(nombre);  out.writeInt(conv);  out.writeDouble(nota);
      out.close();
      ObjectInputStream in = new ObjectInputStream(new FileInputStream(new File(fichero)));
      System.out.println("Valor leído de nombre: " + in.readUTF());
      System.out.println("Valor leído de convocatoria: " + in.readInt());
      System.out.println("Valor leído de nota: " + in.readDouble());
     in.close();
   } catch (FileNotFoundException e) {
         System.err.println("Problemas con el fichero " + fichero + "." + e.getMessage());
   } catch (IOException e) {
         System.err.println("Problemas al escribir en el fichero " + fichero);
}}}
```

Ejemplo de uso de Ficheros Binarios Secuenciales de Objetos (I)

- Como ejemplo, se va a utilizar la escritura y lectura de objetos en un fichero para almacenar y recuperar los valores de una agenda.
- Para el ejemplo, se utilizarán las clases ya vistas:
  - ItemAgenda (información individual de un contacto en la Agenda),
  - Agenda (información de todos los contactos),

```
import java.io.*;

public class ItemAgenda implements Serializable {

   private String nom; private String tel; private int postal;

   public ItemAgenda(String n, String t, int p) { nom = n; tel = t; postal = p; }
   public String toString() { return nom +": "+tel+" ("+postal+")"; }

   // otros métodos ...

} // fin de la clase ItemAgenda
```

Ejemplo de uso de Ficheros Binarios Secuenciales de Objetos (II)

- Recuérdese que es necesario decir que los objetos que se van a guardar son "serializables", lo que se hace declarándolo en las cabeceras de las clases.

```
import java.io.*;

public class Agenda implements Serializable {

   public static final int MAX = 8;

   private ItemAgenda[] elArray;  private int num;

   public Agenda() { elArray = new ItemAgenda[MAX];  num = 0; }

   public void insertar(ItemAgenda b) {
      if (num >= elArray.length) duplicaEspacio();  elArray[num++]=b;  }

   public String toString() {
      String res="";  for (int i=0;i<num;i++)  res += elArray[i]+"\n";
      res += "================================";
      return res; }

  // Otros métodos de la clase ... como eliminar, recuperarPorNombre, …, etc.
```

Ejemplo de uso de Ficheros Binarios Secuenciales de Objetos (III)

- En la clase Agenda se definen **además** métodos para escribir y leer un objeto Agenda:

```
// en la clase Agenda …
public void guardarAgenda(String fichero) {
      try {
        FileOutputStream fos = new FileOutputStream(fichero);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(this); oos.close();
      } catch (IOException ex) { System.err.println("Error al guardar: " + ex.getMessage()); }   }

 public static Agenda leerAgenda(String fichero) {
      Agenda aux = null;
      try {
        FileInputStream fis = new FileInputStream(fichero);
        ObjectInputStream ois = new ObjectInputStream(fis);
        aux = (Agenda)ois.readObject(); ois.close();
      } catch (IOException ex1)  {System.err.println("Error al recuperar: " + ex1.getMessage()); }
        catch (ClassCastException ex2) {System.err.println("Error de tipo: " + ex2.getMessage()); }
      return aux;   }
} // fin de la clase Agenda
```

Ejemplo de uso de Ficheros Binarios Secuenciales de Objetos (IV)

- En la clase GestorPrueba se crea, guarda y recupera una Agenda con algunos elementos:

```
import java.io.*;

public class GestorPrueba {

   public static void main(String[] args) {
      ItemAgenda i1 = new ItemAgenda("Enrique Perez","622115611",46022);
      ItemAgenda i2 = new ItemAgenda("Rosalía","963221153",46010);
      ItemAgenda i3 = new ItemAgenda("Juan Duato","913651228",18011);

      Agenda a1 = new Agenda();
      a1.insertar(i1); a1.insertar(i2); a1.insertar(i3);

      // Guardar y mostrar la Agenda a1:
      a1.guardarAgenda("agenda1.dat");
      System.out.println("AGENDA ALMACENADA:");    System.out.println(a1);

      // Leer del fichero y mostrar la Agenda leída:
      Agenda rec = Agenda.leerAgenda("agenda1.dat");
      System.out.println("AGENDA RECUPERADA:");    System.out.println(rec);
   }
} // fin de la clase GestorPrueba
```

Ejemplo de uso de Ficheros Binarios Secuenciales de Objetos (y V)

- Cuando se ejecuta el main de GestorPrueba se obtiene lo siguiente:

```
AGENDA ALMACENADA:
Enrique Perez: 622115611 (46022)
Rosalía: 963221153 (46010)
Juan Duato: 913651228 (18011)
================================
AGENDA RECUPERADA:
Enrique Perez: 622115611 (46022)
Rosalía: 963221153 (46010)
Juan Duato: 913651228 (18011)
================================
```

- Y en el directorio de trabajo se tiene el fichero:

```
$ ls –l *.dat
-rw-r--r-- 1 profesor PRG  276 2012-03-20 18:00 agenda1.dat
```

Determinación del final del fichero

- A veces no se conoce inicialmente el número de elementos que contiene un fichero.
- Si se intenta leer más allá del **final** del mismo se producirá una excepción (*IOException*) que permitirá, si es tratada, gestionar la situación.
- Además, en el caso de los InputObjectStream ocurre que cuando se leen valores elementales se produce, al intentar acceder más allá del final del fichero, la excepción *EOFException* (subclase de IOException).
- **En conclusión:** es posible tratar todos los elementos de un fichero leyéndolos uno a uno hasta que se produce la excepción correspondiente.
- El ejemplo siguiente muestra esta técnica que, además, se puede utilizar en cualquier flujo que lanze una excepción similar.

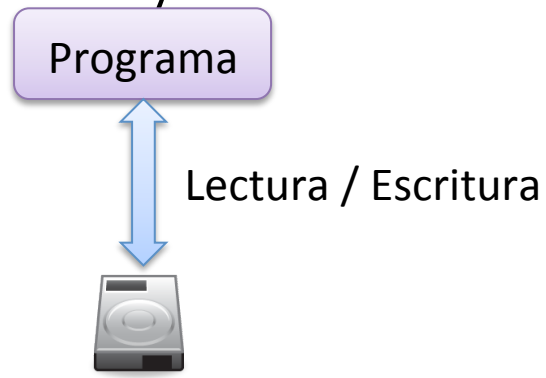
Determinación del final del fichero

- El ejemplo siguiente muestra la lectura de un fichero de int para el que no es conocido el número de valores que contiene.
- Los valores leídos se escriben una tras otro en la pantalla y al llegar al final del fichero se escribe un aviso.

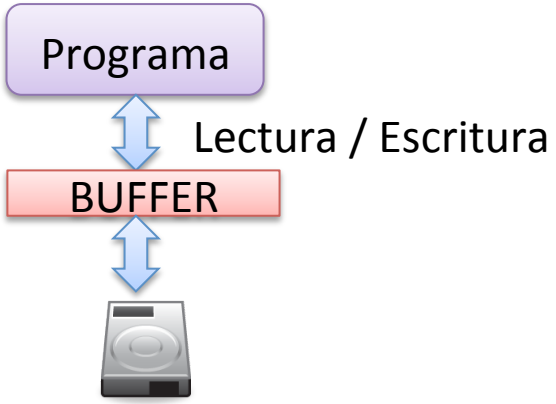
```
public static void leer(String fich) {
    try {
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fich));

        try {
            while (true) {
                int val = ois.readInt();
                System.out.println(val);
            }
        } catch (EOFException ef) { System.out.println("Final del fichero");  }

        ois.close();
    } catch (IOException fex) { System.err.println("Error al recuperar: " + fex.getMessage()); }
}
```

Lectura y Escritura Eficiente de Ficheros Binarios (I)

- En principio, las escrituras de datos mediante ObjectOutputStream (al escribir sobre un FileOutputStream) desencadenan escrituras inmediatas sobre el disco, un proceso bastante costoso.
- Solución eficiente: Uso de un *buffer* que almacena en memoria temporalmente los datos a guardar y, cuando hay suficientes datos, se desencadena la escritura en disco.
  - Es más eficiente realizar 1 escritura en disco de 100 Kbytes que 100 escrituras de 1 Kbyte.



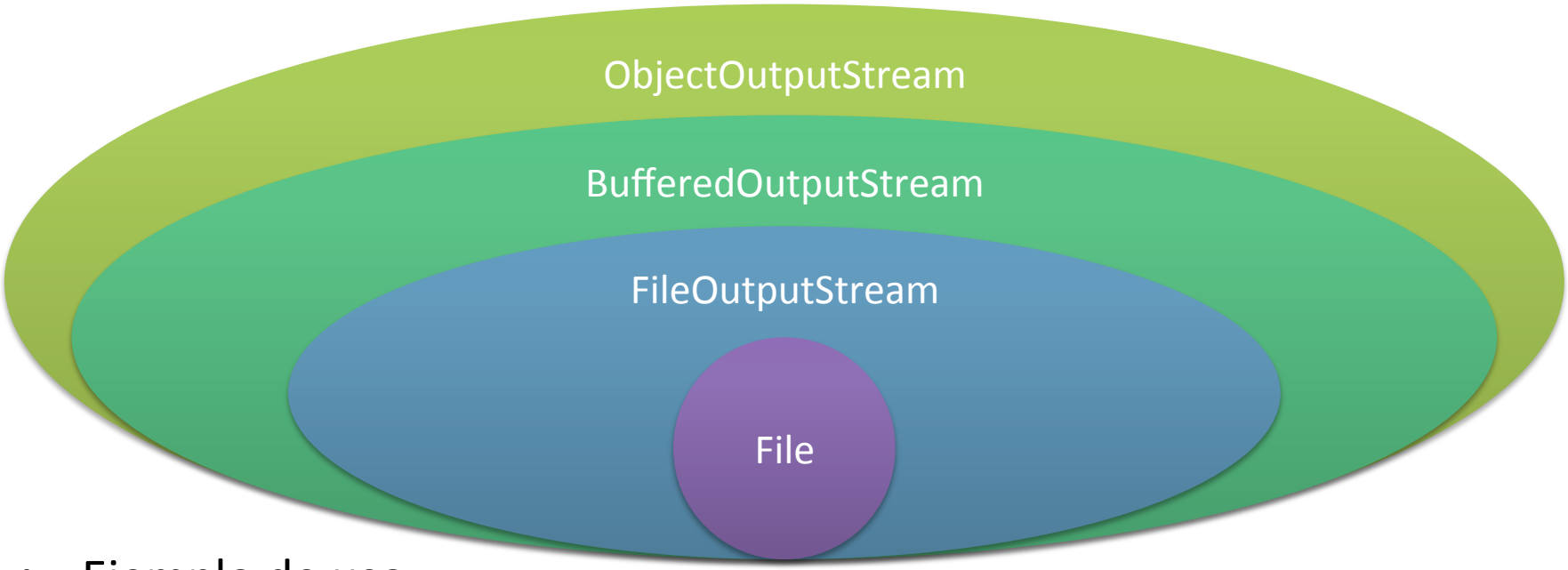
E/S sin buffering



E/S con buffering

Lectura y Escritura Eficiente de Ficheros Binarios (II)

- En Java, la lectura/escritura con *buffering* se realiza mediante las clases BufferedInputStream y BufferedOutputStream.



- Ejemplo de uso:
  - `ObjectOutputStream out = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(new File(fichero))));`
  - `out.writeInt(45);`

Recomendaciones al Trabajar con Ficheros

- Los ficheros (streams) siempre han de ser explícitamente cerrados (método *close*) después de ser utilizados.
  - En particular, un fichero sobre el que se ha escrito, debe ser cerrado antes de poder ser abierto para lectura (para garantizar la escritura de datos en el disco).
- Si el programador no cierra de forma explícita el fichero, Java lo hará, pero:
  - Si el programa termina anormalmente (se va la luz!) y el stream de salida usaba buffering, el fichero puede estar incompleto o corrupto.
- Es importante gestionar las excepciones en los procesos de E/S:
  - Ficheros inexistentes (*FileNotFoundException*).
  - Fallos de E/S (*IOException*).
  - Incoherencias de tipos al usar Scanner (*InputMismatchException*).
  - Condición de final de fichero (*EOFException*).

Conclusiones

- El uso de ficheros permite que la información sobreviva a la ejecución de los programas mediante el uso de almacenamiento secundario.
- Java permite la creación de ficheros binarios independientes de la plataforma y ficheros de texto usando mecanismos basados en flujos (streams).
- La forma más cómoda para interactuar con ficheros de texto:
  - PrintWriter (escritura), Scanner (lectura).
- La forma más cómoda para interactuar con ficheros binarios secuenciales de valores elementales y/o objetos:
  - ObjectOutputStream (escritura), ObjectInputStream (lectura)