



UD6

LENGUAJE DE PROGRAMACIÓN JAVA

6.3 Ficheros serializables

Introducción

Con esta unidad perseguimos los siguientes objetivos:

- ⊕ Crear clases que implementan la interfaz Serializable.
- ⊕ Guardar objetos en disco utilizando la clase ObjectOutputStream como flujo de escritura de objetos.
- ⊕ Recuperar objetos guardados en disco utilizando la clase ObjectInputStream como flujo de lectura de objetos.

La interfaz Serializable

La interfaz Serializable aporta a las clases que la implementan la capacidad de persistencia de sus objetos, persistir significa "durar a lo largo del tiempo".

Hasta ahora, cuando creamos un objeto en un programa, el objeto solo dura lo que dura la ejecución de dicho programa, incluso menos, ya que, si el objeto se ha declarado dentro de un método, su duración será el tiempo de ejecución del método, no del programa. Si una clase implementa la interfaz Serializable, los objetos que pertenecen a dicha clase podrán ser grabados en disco y recuperados las veces que sea necesario, incluso en distintos programas.

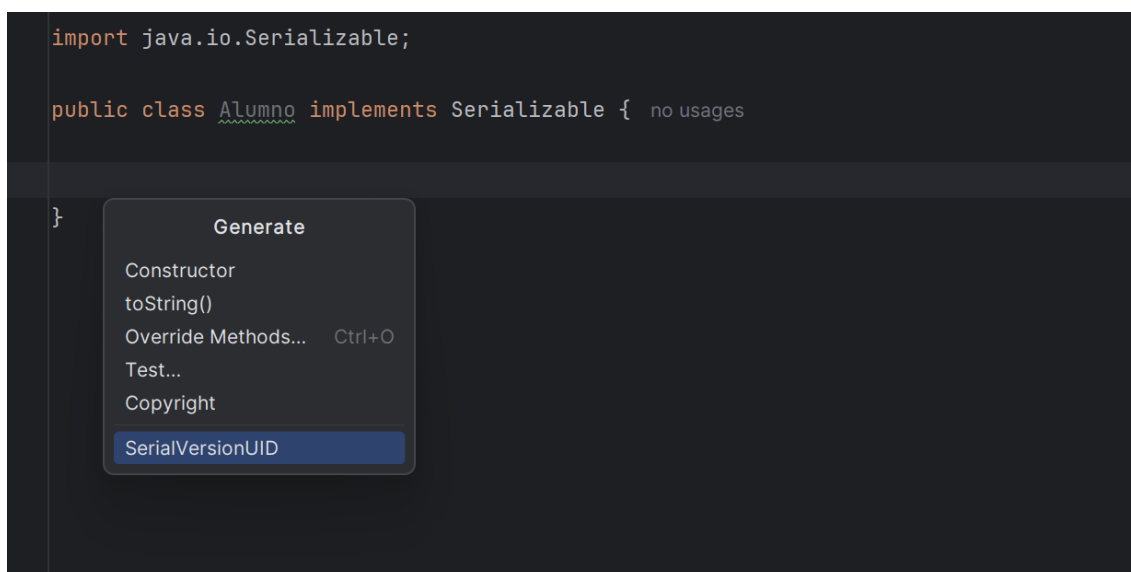
Vamos a crear una clase llamada Alumno con capacidad de persistencia. Para ello crea un nuevo proyecto en el IDE con el nombre que desees y comienza por crear la clase Alumno, de momento vacía como ves a continuación:

```
import java.io.Serializable;

public class Alumno implements Serializable {

}
```

Si estamos usando **IntelliJ o Visual Studio Code** debemos de instalar una extensión o plugin para poder generar los serialVersionUID. Una vez instalado daremos a la opción de generar código y escogeremos la opción de serialVersionUID, como se puede ver en la siguiente figura.



¿Qué es serialVersionUID?

La serialVersionUID es el número de versión de la clase, y se utiliza para evitar problemas de incompatibilidad de versión en los procesos de serialización y deserialización entre los programas que hacen de emisor y receptor del objeto. ¿Y cuándo se produce un problema de incompatibilidad?

Lo comprenderás mejor con un ejemplo:

Tenemos un programa A con una clase Alumno que cuenta con las propiedades nombre, edad y teléfono. Dentro de la clase Principal creamos un objeto Alumno y lo guardamos en un archivo llamado datos.dat. El programa A es el que realiza la serialización y por lo tanto el emisor del objeto guardado.

Tenemos otro programa B, donde hemos copiado la clase Alumno, pero esta vez se nos ha ocurrido añadir una propiedad más llamada domicilio. Los programas A y B tienen distinta versión de la clase Alumno.

Ahora desde la clase Principal del programa B recuperamos el objeto Alumno que previamente guardamos en el archivo datos.dat durante la ejecución del programa A.

El programa B debe realizar la deserialización del objeto guardado y por lo tanto será el receptor de dicho objeto. El programa B nos arroja una excepción, ya que intenta recuperar un objeto construido a partir de la primera versión de la clase Alumno, sin embargo, el programa B contiene la segunda versión de la clase Alumno, que resulta incompatible.

Las versiones primera y segunda de la clase Alumno deberían tener distinto serialVersionUID para poder distinguir rápidamente que se trata de versiones distintas de la misma clase. De esta forma, en el proceso de deserialización, la máquina virtual de Java comparará la serialVersionUID del objeto guardado con la serialVersionUID de la clase Alumno que contiene el programa B, arrojando una excepción de tipo `InvalidClassException`.

¿Y qué pasa si no declaramos la serialVersionUID? Si dentro de la clase no hemos especificado un valor de serialVersionUID, la máquina virtual de Java debe examinar las clases de origen y destino generando las serialVersionUID de forma dinámica. Aunque no sea obligatorio, resulta mucho más rápido y efectivo declarar la serialVersionUID y modificarla en cada nueva versión.

Vamos a crear una clase llamada Alumno con capacidad de persistencia. Para ello crea un nuevo proyecto en nuestro IDE con el nombre que desees y comienza por crear la clase Alumno, generamos el serialVersiobUID con el plugging que habíamos instalado y le añadimos la notación de `@Serial`:

```
import java.io.Serial;
import java.io.Serializable;

public class Alumno implements Serializable { no usages

    @Serial no usages
    private static final long serialVersionUID = -29358239215567888L;
}
```

Si borramos la línea del serialVersion y volvemos a realizar la misma operación, generará exactamente el mismo número. Sin embargo, si modificas algo, por ejemplo, añadiendo dos propiedades, generará distinto número. Cambia ahora la clase Alumno dejándola así:

```
public class Alumno implements Serializable { no usages

    @Serial private static final long serialVersionUID = 6573908600555625686L; no usages

    private String nombre; 2 usages
    private int edad; 2 usages

    public Alumno(String nombre, int edad) { no usages
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() { no usages
        return this.nombre;
    }

    public int getEdad() { no usages
        return this.edad;
    }
}
```

Grabar un objeto en disco

En este apartado veremos cómo grabar un objeto de la clase Alumno en un archivo. Para empezar, debes completar el código de la clase Alumno y de la clase Calificacion con el siguiente código:

```
import java.io.Serial;
import java.io.Serializable;

public class Calificacion implements Serializable { 3 usages
    @Serial no usages
    private static final long serialVersionUID = -7942100049483722991L;
    private String asignatura; 2 usages
    private int nota; // Sobre 100 2 usages

    public Calificacion(String asignatura, int nota) { 1 usage
        this.asignatura = asignatura;
        this.nota = nota;
    }

    @Override
    public String toString() {
        return "Calificacion{" +
            "asignatura='" + asignatura + '\'' +
            ", nota=" + nota +
            '}';
    }
}
```

```
import java.io.Serial;
import java.io.Serializable;
import java.util.LinkedList;
import java.util.List;

public class Alumno implements Serializable { no usages

    @Serial no usages
    private static final long serialVersionUID = 8174942064592305815L;
    private String nombre; 2 usages
    private int edad; 2 usages
    private LinkedList<Calificacion> calificaciones; 3 usages

    public Alumno(String nombre, int edad) { no usages
        this.nombre = nombre;
        this.edad = edad;
        this.calificaciones = new LinkedList<>();
    }

    public void calificar(String asignatura, int nota){ no usages
        this.calificaciones.add(new Calificacion (asignatura, nota));
    }

    public String getNombre() { no usages
        return this.nombre;
    }

    public int getEdad() { no usages
        return this.edad;
    }

    public List<Calificacion> getCalificaciones(){ no usages
        return this.calificaciones;
    }
}
```

De esta forma ya tenemos todo listo para construir un objeto Alumno y persistirlo grabándolo en un archivo.

```

3  import java.io.FileOutputStream;
4  import java.io.IOException;
5  import java.io.ObjectOutputStream;
6
7  public class Principal {
8      public static void main(String [] args){
9          Alumno alu1 = new Alumno( nombre: "Pedro", edad: 25);
10         alu1.calificar( asignatura: "Matemáticas", nota: 50);
11         alu1.calificar( asignatura: "Inglés", nota: 75);
12         alu1.calificar( asignatura: "Informática", nota: 95);
13         alu1.calificar( asignatura: "Lengua", nota: 60);
14
15         // Abrir fichero para escritura
16         FileOutputStream file;
17         ObjectOutputStream buffer;
18         try {
19             file = new FileOutputStream( name: "C:\\ficherosTests\\alumno.dat");
20             buffer = new ObjectOutputStream(file);
21         } catch (IOException e) {
22             System.out.println("No se ha podido abrir el fichero");
23             System.out.println(e.getMessage());
24             return;
25         }
26
27         // Guarda objeto en el fichero alumno.dat
28         try {
29             buffer.writeObject(alu1);
30             System.out.println("El objeto se ha grabado con éxito");
31         } catch (IOException e) {
32             System.out.println("Error al escribir en el fichero");
33             System.out.println(e.getMessage());
34         }
35
36         // Cerrar el fichero
37         try {
38             buffer.close();
39             file.close();
40         } catch (IOException e) {
41             System.out.println("Error al cerrar el fichero");
42             System.out.println(e.getMessage());
43         }
44     }
45 }

```

Vamos a analizar el código anterior:

En primer lugar, hemos creado un objeto de la clase Alumno llamado alu1 y le hemos añadido cuatro calificaciones.

```
File = new FileOutputStream("C:\\FicherosTest\\alumno.dat");  
buffer = new ObjectOutputStream(File);
```

Después hemos construido un objeto de la clase `FileOutputStream` (iniciador) dejando el archivo `alumno.dat` abierto para escritura. El fichero será creado en cada ejecución, sobrescribiendo la versión anterior si existe. También podemos abrir el archivo para añadir sin eliminar los datos anteriores; en ese caso tendríamos que establecer a `true` el segundo argumento:

```
File = new FileOutputStream("C:\\FicherosTest\\alumno.dat", true);
```

Ya abierto el fichero, creamos un objeto `ObjectOutputStream` que nos servirá como filtro para mejorar el proceso de escritura.

```
buffer.writeObject(alu1);
```

El método `writeObject()` de la clase `ObjectOutputStream` es el que nos permite grabar el objeto en disco.

```
buffer.close();  
file.close();
```

Por último, igual que en todas las operaciones de entrada / salida, hay que terminar cerrando los flujos de datos, liberando así el fichero.

Recuperar un objeto desde disco

Ha llegado el momento de recuperar el objeto `Alumno` guardado en disco. Crearemos un nuevo proyecto, pero necesitarás la implementación de las clases `Alumno` y `Calificacion`, así que puedes copiarlas del proyecto anterior.

Vamos a seguir los tres pasos habituales: abrir, leer, cerrar. Esta vez hemos utilizado los flujos de lectura `FileInputStream` (iniciador) y `ObjectInputStream` (filtro).

```

import java.io.*;

public class Principal {
    public static void main(String[] args) {

        // Abrir fichero para escritura
        FileInputStream file;
        ObjectInputStream buffer;
        try {
            file = new FileInputStream("C:\\ficherosTests\\alumno.dat");
            buffer = new ObjectInputStream(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Lee el objeto guardado en el archivo alumno.dat
        try {
            Alumno alu1 = (Alumno) buffer.readObject();
            System.out.println("Nombre del alumno: " + alu1.getNombre());
            System.out.println("Edad: " + alu1.getEdad());
            for (Calificacion c : alu1.getCalificaciones()) {
                System.out.println(c);
            }
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }
    }
}

```

Con la sentencia `Alumno alu1 = (Alumno) buffer.readObject();` hemos leído el objeto, pero hemos tenido que convertirlo a tipo `Alumno`, ya que el método `readObject()` devuelve un genérico `Object`.

El modificador transient

El modificador `transient` se utiliza con clases serializables para indicar las propiedades que no queremos que sean serializadas, es decir, las que no deseamos que se guarden. Tiene sentido con algunas propiedades, tales como un password.

Para utilizarlo simplemente debemos de añadir el modificador `transient` a la propiedad `edad` de la clase `Alumno`.

```
private transient int edad;
```

Grabar y recuperar varios objetos

En este apartado veremos cómo podemos guardar en un Fichero varios objetos del mismo tipo y cómo podemos posteriormente leerlos controlando el final del fichero.

Como ejemplo, vamos a crear una agenda de contactos. En primer lugar, debes crear un nuevo proyecto en el IDE y una clase `Contacto`, que representará a cada uno de los contactos que queremos guardar en la agenda.

```
public class Contacto implements Serializable { 6 usages
    @Serial no usages
    private static final long serialVersionUID = -4624046047796483183L;

    private String nombre; 3 usages
    private String telefono; 3 usages

    public Contacto(String nombre, String telefono) { 3 usages
        this.nombre = nombre;
        this.telefono = telefono;
    }

    public String getNombre() { no usages
        return nombre;
    }

    public String getTelefono() { no usages
        return telefono;
    }

    @Override
    public String toString() {
        return "Contacto [" + nombre + " - " + telefono + "];"
    }
}
```

Ahora, en la clase Principal, guardaremos tres contactos en el fichero agenda.dat.

```
public class Main {
    public static void main(String[] args) {

        FileOutputStream file;
        ObjectOutputStream buffer;
        try {
            file = new FileOutputStream( name: ".\\src\\resources\\agenda.dat", append: true);
            buffer = new ObjectOutputStream(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir el fichero");
            System.out.println(e.getMessage());
            return;
        }

        // Creamos tres contactos
        Contacto c1 = new Contacto( nombre: "Amelia", telefono: "913670542");
        Contacto c2 = new Contacto( nombre: "Federico", telefono: "6166644422");
        Contacto c3 = new Contacto( nombre: "Carmen", telefono: "639888777");

        // Creamos tres contactos
        Contacto c1 = new Contacto( nombre: "Amelia", telefono: "913670542");
        Contacto c2 = new Contacto( nombre: "Federico", telefono: "6166644422");
        Contacto c3 = new Contacto( nombre: "Carmen", telefono: "639888777");

        // Guardamos los tres contactos en agenda.dat
        try {
            buffer.writeObject(c1);
            buffer.writeObject(c2);
            buffer.writeObject(c3);
            System.out.println("Los contactos se han guardado correctamente");
        } catch (IOException e) {
            System.out.println("Error al escribir en el fichero");
            System.out.println(e.getMessage());
        }

        // Cerrar el fichero
        try {
            buffer.close();
            file.close();
        } catch (IOException e) {
            System.out.println("Error al cerrar el fichero");
            System.out.println(e.getMessage());
        }

    }
}
```

Leer contactos hasta que sea final de fichero

Ahora vamos a realizar un listado de contactos, para lo cual debes crear un nuevo proyecto con el nombre que desees y copiar la clase Contacto del proyecto anterior. Para leer cada contacto debes utilizar el método `readObject()`, tal como ya has aprendido, pero en esta ocasión hay varios objetos de la clase Contacto para leer dentro del fichero, con lo cual necesitarás iterar, es decir, utilizar un bucle para leer sucesivas veces mientras no sea final de fichero.

Cuando intentamos leer un objeto del flujo de lectura y ya no hay más objetos para leer se produce una excepción de tipo `EOFException`. En el programa que sigue estamos capturando la excepción `EOFException` cuando se produce y realizando la siguiente asignación: `eof = true`. De esta forma podemos leer objetos mientras la variable `eof` se mantenga con el valor `false`.

```
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Principal {
    public static void main(String[] args) {
        // Abrimos fichero agenda.dat para lectura
        FileInputStream file;
        ObjectInputStream buffer;

        try {
            file = new FileInputStream("src\\resources\\agenda.dat");
            buffer = new ObjectInputStream(file);
        } catch (IOException e) {
            System.out.println("No se ha podido abrir la agenda");
            System.out.println(e.getMessage());
            return;
        }

        // Leemos la lista de contactos
        boolean eof = false;
        Contacto c;
        while (!eof) {
            try {
                c = (Contacto) buffer.readObject();
                System.out.println(c);
            } catch (EOFException e1) {
                eof = true;
            } catch (IOException e2) {
                System.out.println("Error al leer los contactos");
                System.out.println(e2.getMessage());
            } catch (ClassNotFoundException e3) {
                System.out.println("La clase Contacto no es correcta");
                System.out.println(e3.getMessage());
            }
        }
    }
}
```

```
// Cerramos el fichero
try {
    buffer.close();
    file.close();
} catch (IOException e) {
    System.out.println("Error al cerrar el fichero");
    System.out.println(e.getMessage());
}
}
```

Controlar el final de fichero con el método available()

Otro sistema para controlar cuándo hemos llegado a final de fichero es mediante el método `available()` de la clase `FileInputStream`, que devuelve un número entero con el número de bytes pendientes de lectura. En el siguiente programa estamos asignando a la variable `bytesEnBuffer` el valor devuelto por el método `available()` antes de comenzar la lectura y después de leer cada objeto. De esta forma podemos leer mientras se cumpla la siguiente condición: `bytesEnBuffer > 0`.

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Principal {
    public static void main(String[] args) {
        int bytesEnBuffer;

        // Abrimos fichero agenda.dat para lectura
        FileInputStream file;
        ObjectInputStream buffer;
        try {
            file = new FileInputStream("name: ".\\"src\\resources\\agenda.dat");
            buffer = new ObjectInputStream(file);
            bytesEnBuffer = file.available();

        } catch (IOException e) {
            System.out.println("No se ha podido abrir la agenda");
            System.out.println(e.getMessage());
            return;
        }
    }
}
```

```
// Leemos la lista de contactos
System.out.println("Bytes por leer: " + bytesEnBuffer);
Contacto c;
while (bytesEnBuffer > 0) {
    try {
        c = (Contacto) buffer.readObject();
        System.out.println(c);
        bytesEnBuffer = file.available();
        System.out.println("Bytes pendientes en buffer: " + bytesEnBuffer);
    } catch (IOException e2) {
        System.out.println("Error al leer los contactos");
        System.out.println(e2.getMessage());
    } catch (ClassNotFoundException e3) {
        System.out.println("La clase Contacto no es correcta");
        System.out.println(e3.getMessage());
    }
}

// Cerramos el fichero
try {
    buffer.close();
    file.close();
} catch (IOException e) {
    System.out.println("Error al cerrar el fichero");
    System.out.println(e.getMessage());
}
}
```

Referencias

Apuntes elaborados a partir de la siguiente documentación:

- [1] Apuntes Fernando Barber y Ricardo Ferris. Universidad de Valencia.
- [2] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.
- [3] Apuntes de Programación de Carlos Cacho y Raquel Torres. Ceedcv.
- [4] Apuntes de Programación Edix Digital Workers.

Licencia



[CC BY-NC-SA 3.0 ES](https://creativecommons.org/licenses/by-nc-sa/3.0/es/) Reconocimiento - No Comercial - Compartir Igual (by-nc-sa)

No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

NOTA: Esta es una obra derivada de la original realizada por Carlos Cacho y Raquel Torres.