1.2. Lectura / escritura de objetos

Índice

El contenido de esta lección ya lo estudiaste en la asignatura de Programación, pero te recomendamos que vuelvas a estudiarlo como repaso, ya que te ayudará a continuar con éxito el resto de contenidos y a realizar las actividades propuestas sin dificultad.

Adelante, te costará poco esfuerzo recordar estos conceptos y volverlos a poner en la práctica.

Introducción a la unidad formativa

Objetivos

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.

¡Ánimo y adelante!

Lectura y escritura de objetos

La interfaz Serializable

La interfaz *Serializable* aporta a las clases que la implementan la capacidad de persistencia de sus objetos.

¿Y qué es la persistencia?

Según el diccionario de la Real Academia Española, persistir significa "durar a lo largo del tiempo".

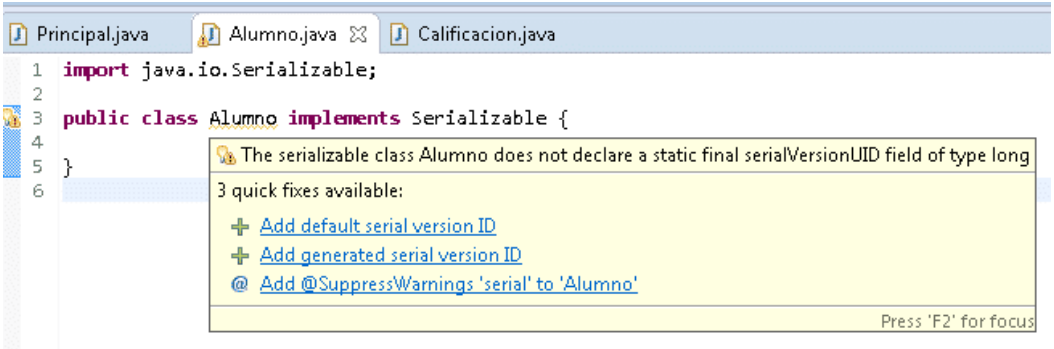
Hasta ahora, cuando creamos un objeto en un programa, el objeto solo dura lo que dura la ejecución de dicho programa, incluso menos, ya que si el objeto se ha declarado dentro de un método, su duración será el tiempo de ejecución del método, no del programa.

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 Eclipse 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 ya has creado la clase, comprobarás que Eclipse te está dando una alerta, la palabra *Alumno* está subrayada en amarillo y, si sitúas el puntero de ratón durante un rato sobre la palabra *Alumno*, obtendrás un cuadro con información sobre la alerta y las soluciones propuestas por Eclipse.



La alerta nos dice lo siguiente: "*The serializable class Alumno does not declare a static final serialVersionUID fields of type long*". Está claro que nos está pidiendo que declaremos una variable llamada *serialVersionUID*.

¿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 *telefono*. 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 está 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 un 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.

Volvamos a poner de nuevo la atención en la clase Alumno.

Vuelve a situar el puntero del ratón sobre el nombre de la clase hasta que salga el cuadro informativo y selecciona la opción "*add generated serial version ID*". Verás que se añade automáticamente la variable *serialVersionUID* con un valor auto calculado.

```
import java.io.Serializable;

public class Alumno implements Serializable {
	private static final long serialVersionUID = 4854486451470258537L;

}
```

Verás que si borras la nueva línea para que vuelva a salir la alerta y vuelves a realizar de nuevo la 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:

```
import java.io.Serializable;

public class Alumno implements Serializable {

	private String nombre;
	private int edad;

}
```

Ahora vuelve a realizar la operación anterior para que vuelva a generarse la *serialVersionUID.*

```
import java.io.Serializable;

public class Alumno implements Serializable {

	private static final long serialVersionUID = 1742837368213302555L;
	private String nombre;
	private int edad;

}
```

El número de versión se genera a partir de un complejo algoritmo que tiene que ver con el contenido de la clase y es susceptible a los cambios que realicemos dentro del código.

En los siguientes apartados completaremos la clase *Alumno* y comenzaremos a realizar operaciones de lectura y escritura de objetos.

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* copiando y pegando el siguiente código:

```
import java.io.Serializable;
import java.util.ArrayList;

public class Alumno implements Serializable {
	private static final long serialVersionUID = 4854486451470258537L;

	private String nombre;
	private int edad;
	private ArrayList<Calificacion> calificaciones;

	public Alumno(String nombre, int edad) {
		this.nombre = nombre;
		this.edad = edad;
		this.calificaciones = new ArrayList<Calificacion>();
	}

	public void calificar(String asignatura, int nota) {
		this.calificaciones.add(new Calificacion(asignatura, nota));
	}

	public String getNombre() {
		return nombre;
	}
	public int getEdad() {
		return edad;
	}
	public ArrayList<Calificacion> getCalificaciones() {
		return calificaciones;
	}
}
```

Como puedes comprobar por el código, un objeto *Alumno* está formado por las propiedades *nombre*, *edad* y una colección de objetos *Calificacion*.

La clase *Calificacion* tiene la siguiente implementación:

```
import java.io.Serializable;

public class Calificacion implements Serializable {
	private static final long serialVersionUID = 3057545624874202352L;

	private String asignatura;
	private int nota; // Sobre 100
```

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

	@Override
	public String toString() {
		return "Calificación [Asignatura=" + asignatura + ", Nota=" + nota +
"]";
	}
}
```

Ya tenemos todo listo para construir un objeto *Alumno* y persistirlo grabándolo en un archivo.

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class Principal {
	public static void main(String args[]) {
		// Crear objeto Alumno
		Alumno alu1 = new Alumno("Pedro", 25);
		alu1.calificar("Matemáticas", 50);
		alu1.calificar("Inglés", 75);
		alu1.calificar("Informática", 95);
		alu1.calificar("Lengua", 60);

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

		// Guarda objeto en el fichero alumno.dat
		try {
			buffer.writeObject(alu1);
			System.out.println("El objeto se ha grabado con éxito");
		} 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());
            }

        }
}
```

Antes de ejecutar el programa vamos a analizarlo un poco.

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

- ***file = new FileOutputStream("J:\\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("J:\\alumno.dat", true);*

  En el ejemplo, el archivo se guardará en una memoria USB que se encuentra en la unidad J. Cambia la ruta con la ubicación que desees.

  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.

Ahora ya puedes ejecutar el programa.

Si todo ha salido bien, habrás obtenido en pantalla el mensaje "El objeto se ha grabado con éxito". Comprueba con ayuda del explorador de Windows si el archivo se ha creado.

Recuperar un objeto de disco

Ha llegado el momento de recuperar el objeto *Alumno* guardado en disco.

Comienza por crear un nuevo proyecto, pero necesitarás la implementación de las clases *Alumno* y *Calificacion*, así que puedes copiarlas del proyecto anterior.

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

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

            // Abrir fichero para lectura
            FileInputStream file;
            ObjectInputStream buffer;
            try {
                  file = new FileInputStream("J:\\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 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());
            }

      }
}
```

Como has podido comprobar, hemos seguidos los tres pasos habituales: abrir, leer, cerrar. Esta vez hemos utilizado los flujos de lectura *FileInputStream* (iniciador) y *ObjectInputStream* (filtro).

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 ponerlo en práctica sigue estos pasos

1. Añade el modificador *transient* a la propiedad *edad* de la clase *Alumno*.

```
private transient int edad;
```

2. Vuelve a ejecutar el programa creado en el apartado "Grabar un objeto en disco". A pesar de que construimos un objeto *Alumno* con *nombre* "Pedro" y *edad* 25 años, la *edad* no se habrá guardado.

3. Ahora vuelve a ejecutar el programa creado en el apartado "Recuperar un objeto de disco". El resultado es el siguiente:

   Nombre del alumno: Pedro
   Edad: 0
   Calificación [Asignatura=Matemáticas, Nota=50]
   Calificación [Asignatura=Inglés, Nota=75]
   Calificación [Asignatura=Informática, Nota=95]
   Calificación [Asignatura=Lengua, Nota=60]

Como la *edad* no se guardó, el objeto recuperado tiene el valor por defecto para una variable de tipo *int*, es decir, un cero.

Grabar y recuperar varios objetos

En este apartado veremos cómo podemos guardar en un fichero varios objetos del mismo tipo 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 Eclipse y una clase *Contacto*, que representará a cada uno de los contactos que queremos guardar en la agenda.

```
import java.io.Serializable;

public class Contacto implements Serializable {
	private static final long serialVersionUID = -4624046047796483183L;

	private String nombre;
	private String telefono;

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

	public String getNombre() {
		return nombre;
	}
	public String getTelefono() {
		return telefono;
	}

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

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

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class Principal {

	public static void main(String[] args) {
		// Abrimos fichero para escritura
		FileOutputStream file;
		ObjectOutputStream buffer;
		try {
			file = new FileOutputStream("D:\\agenda.dat", 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("Amelia", "913670542");
            Contacto c2 = new Contacto("Federico", "6166644422");
            Contacto c3 = new Contacto("Carmen", "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 con éxito");
            } 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.

¿Pero cómo podemos saber cuándo se ha llegado al final de fichero?
La solución está en la excepción *EOFException*

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.ObjectInputStream;
import java.io.IOException;

public class Principal {
```

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

            // Abrimos fichero agenda.dat para lectura
            FileInputStream file;
            ObjectInputStream buffer;
            try {
                  file = new FileInputStream("D:\\agenda.dat");
                  buffer = new ObjectInputStream(file);
            } catch (IOException e) {
                  System.out.println("No se ha podido abrir la agenda de
contactos");
                  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 de la
agenda");
                        System.out.println(e2.getMessage());
                  } catch (ClassNotFoundException e3) {
                        System.out.println("La clase Contacto no está cargada en
memoria");
                        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.ObjectInputStream;
import java.io.IOException;

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

            // Abrimos fichero agenda.dat para lectura
            FileInputStream file;
            ObjectInputStream buffer;
            try {
                  file = new FileInputStream("D:\\agenda.dat");
                  buffer = new ObjectInputStream(file);
                  bytesEnBuffer = file.available();
            } catch (IOException e) {
                  System.out.println("No se ha podido abrir la agenda de
contactos");
                  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);
                  } catch (IOException e2) {
                        System.out.println("Error al leer los contactos de la
agenda");
                        System.out.println(e2.getMessage());
                  } catch (ClassNotFoundException e3) {
                        System.out.println("La clase Contacto no está cargada en
memoria");
                        System.out.println(e3.getMessage());
                  }
                  bytesEnBuffer = file.available();
                  System.out.println("Bytes pendientes en buffer: " +
bytesEnBuffer);
            }

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

      }
}
```

Despedida

Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Para que un objeto tenga la capacidad de persistencia debe implementar la interfaz ***Serializable***.
- Podemos guardar objetos en disco utilizando la clase ***ObjectOutputStream***.
- Podemos recuperar objetos guardados en disco utilizando la clase ***ObjectInputStream***.