Serialización: Transferencia de objetos

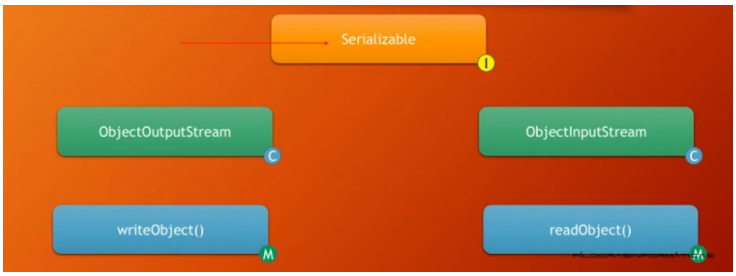
La serialización consiste en convertir un objeto en una serie de bytes , con el objetivo de almacenar ese objeto en un sistema de almacenamiento masivo como un HD y poderlo restaurar o recomponer dicho objeto en un futuro.

Nuestro objetivo más adelante será distribuir un objeto a través de la red a ordenadores remoto y en esos ordenadores remoto ese objeto sea restablecido.

Clases a utilizar:
Para realizar la serialización de un objeto, vamos a tener que usar la interfaz "Serializable" , esta interfaz no tiene métodos, por lo que aquellas clases que implemente esta interfaz no se verá obligada a sobreescribir ningún método, lo único que hay que hacer es indicarle a la clase que implemente esta interfaz.

Esto implica que los objetos que pertenece a la clase que implementa "Serializable" , son susceptible de ser convertidos en bytes para ser transferidos a través de la red o almacenarlo en el HD por ejemplo.

Además de la interfaz, será necesario manejar las clases "ObjecOutputStream" , que construye un flujo de datos por el cual es posible transferir un objeto desde nuestro programa de java hacia un medio de almacenamiento o hacia un equipo remoto y "ObjectImputStream", crea un flujo de datos a través del cual es posible que se transfiera un objeto remoto hacia nuestro programa, para que dentro de nuestro programa recomponer esos bytes para construir el objeto.



Estas clases tienen sus métodos; "writeObject() " que crea esa sucesión de bytes para lanzarlo a través del flujo de datos hacia afuera y "readObject()" que realiza la función inversa, lee esa sucesión de bytes que entra a nuestro programa por el flujo de datos.

Ejemplo 1:

Convertir un objeto java en una sucesión de bytes, guardarlo en el HD del equipo y posteriormente rescatar el objeto del disco duro.

Vamos a tener una clase Empleado cuyos objetos tendrá un nombre, un sueldo y una fecha de contratación

```
class Empleado implements Serializable {

        public Empleado(String n, double s, int año, int mes, int dia){

                nombre=n;
                sueldo=s;
                GregorianCalendar calendario=new GregorianCalendar(año, mes-1,dia);
                fechaContrato=calendario.getTime();

        }
```

También contendrá los siguientes métodos:

```
public String getNombre() //Devuelve el nombre del empleado
```

public double getSueldo() // Devuelve el sueldo delempleado

public Date getFechaContrato() //Devuelve el sueldo del empleado

public void subirSueldo(**double** porcentaje) //Aumenta el sueldo en un porcentaje pasado por parámetro.

public String toString() // Devuelve la cadena : Nombre de Empleado , su sueldo es; "sueldo" y su fecha de contratación es;: "fecha de contratación".

El método toString() pertenece a la clase Object, la clase de la cual heredan todas las clases de java; eso implica que todas las clases de java lleva implícitamente este método por heredar de la clase "Object". El método toString se utiliza para especificar una descripción de nuestra clase, es como pasar a texto que es lo que hace nuestra clase.

Clase Administrador que hereda de Empleado

Atributo : **private double** incentivo

Constructor: **public** Administrador(String n, **double** s, **int** año, **int** mes, **int** dia)

Que llama al constructor de la clase padre y establece "incentivo=0"

Métodos:

public double getSueldo() // devuelve el sueldo base más el incentivo.

public void setIncentivo(**double** b) // establece un incentivo

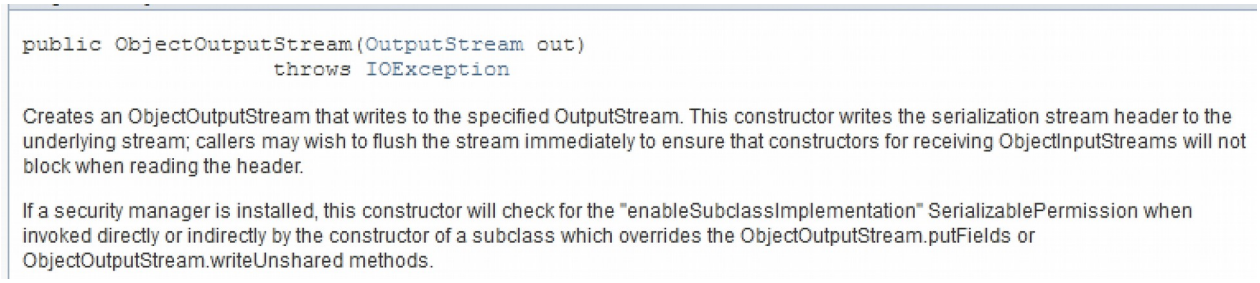
public String toString() // Da una descripción de lo que hace esta clase. Llama al método toString de la clase padre y añade el atributo incentivo.

Clase principal (main).

Establece un objeto de tipo administrador que llamamos jefe , establece un incentivo= 5000 . Creamos un array de Empleado[] , que contiene 3 elementos. En la primera posición almacenamos el objeto administrador y en las dos posiciones restantes almacenamos objetos de tipo empleado pasando los argumentos precisos.

En el ejercicio hay que Serializar el array Empleado; es decir , convertirlo en una sucesión de bytes para almacenarlo en el disco duro y posteriormente hay que rescatarlo del disco e imprimir el resultado en consola.

Vamos a la API y vemos la clase OutputObjectStream, esta clase tiene dos constructores y estos lanza la excepción IOException.



```
public ObjectOutputStream(OutputStream out)
                    throws IOException
```

Creates an ObjectOutputStream that writes to the specified OutputStream. This constructor writes the serialization stream header to the underlying stream; callers may wish to flush the stream immediately to ensure that constructors for receiving ObjectInputStreams will not block when reading the header.

If a security manager is installed, this constructor will check for the "enableSubclassImplementation" SerializablePermission when invoked directly or indirectly by the constructor of a subclass which overrides the ObjectOutputStream.putFields or ObjectOutputStream.writeUnshared methods.

public class Serializando {

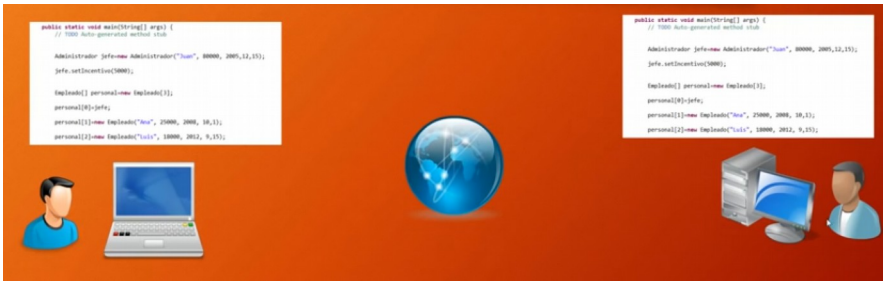
public static void main(String[] args) {

```
        Administrador jefe=new Administrador("Juan", 80000, 2005,12,15);
        jefe.setIncentivo(5000);
        Empleado[] personal=new Empleado[3];
        personal[0]=jefe;
        personal[1]=new Empleado("Ana", 25000, 2008, 10,1);
        personal[2]=new Empleado("Luis", 18000, 2012, 9,15);

/**Construimos un flujo de datos de tipo ObjectOutputStream (flujo de salida).
El constructor de ObjectOutputStream pide un argumento de tipo OutputStream
Le pasamos un objeto FileOutputStrem para especificar donde vamos a escribir esta sucesión de bytes del objeto Empleado*/


/**Le he dado la extensión dat al fichero que se va a crear, la extensión da igual.
Una vez que hemos creado el flujo de dato de salida al exterior,ahora tenemos que escribir ese objeto.
El objeto a escribir es del tipo "Empleado[]" y se llama personal.

 ¿Cuál es el método de ObjectOutputStream que nos permite realizar todo esto?
Visitamos la API y vemos que "writeObject(Object objeto)" escribe un objeto especificado en el ObjectOutputStream */



//Cerramos flujo


/**Una vez que hemos archivado el objeto ahora lo vamos a rescatar y lo vamos a mostrar por pantalla.
Creamos un objeto de tipo "ObjectInputStream" (flujo de entrada) lo llamamos "recupera_fichero"*/



/**Con esto creamos el flujo de entrada.

Ahora usamos el método readObject de la clase ObjectInputStream para leer ese objeto.

Importante: ¿Qué estamos leyendo?. Estamos leyendo un array, al leerlo, necesito almacenarlo en otro array por supuesto de tipo
Empleado

Si consultamos la API, vemos que el métode readObjet lee un objeto desde ObjectinputStream y devuelve un Object
Esto me indica que tengo dos opciones:
1) Me creo un objeto de tipo Object para almacenarlo
2) Si trato un objeto de otro tipo,debo de realizar un casting.*/


// cerramos el flujo ObjectInputStream



//Ahora vamos a imprimir la información de dicho objeto. Lo hacemos mediante un "for-each" que recorra el array recuperado


        }

}
```

SERIALIZACIÓN II : Cambio de versiones (serialVersion UID)

¿Qué sucede cuando cambiamos de versión el programa que se encarga de serializar objetos?; es decir, modificar el programa, mejorar el programa , en definitiva modificar el programa que se encarga de serializar objetos.

¿Qué es y como se utiliza la constante "serialVersionUID?



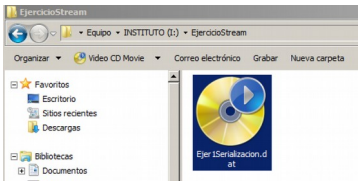
Supongamos el escenario de que un emisor quiere enviar un objeto a un receptor a través de la red. Para que esto sea posible, Tanto el emisor que va a serializar el objeto como el receptor que va a leer el objeto serializado que va a recibir por la red , debe de tener una copia idéntica del programa java (deben tener la misma versión del programa java).

Cuando creamos un programa java, ese programa tiene una huella (número identificativo único denominado SHA, para ese programa que hemos creado).Esta huella la genera el compilador java.

Supongamos que hemos modificado nuestro programa, por ejemplo, hemos añadido un atributo más al objeto. En este caso , cambia la huella del emisor pero no la del receptor. El emisor enviará el objeto serializado pero el receptor no podrá leer dicho objeto.

Ejemplo. Usar el programa del ejercicio anterior.

Supongamos que hemos creado el objeto serializado y lo tenemos en el directorio que hemos creado para el caso.



Ahora comentamos las líneas que serializa el objeto:

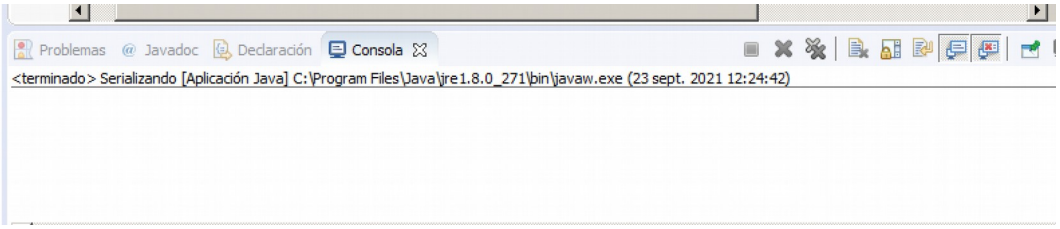
```
//ObjectOutputStream escribiendo_fichero = new ObjectOutputStream(new FileOutputStream("I:\\EjercicioStream\\Ejer1Serializacion.dat"));
//escribiendo_fichero.writeObject(personal);
//escribiendo_fichero.close();
```

Nos ponemos en el caso del receptor y comprobamos que el programa leer el objeto sin problemas y lo imprime.

Descomentar las líneas comentadas anteriormente y borramos el fichero generado del directorio. Modificamos algún campo de la clase empleado, por ejemplo, el campo sueldo lo cambiamos a "sueldos" y ejecutamos el programa generando nuevamente el fichero correspondiente al objeto serializado.

Nos ponemos en la posición del receptor y que este receptor no ha recibido la actualización del campo sueldo, por lo que volvemos a comentar las tres líneas que genera el fichero y cambiamos el campo "sueldos" por sueldo.

Al ejecutar de nuevo el programa vemos que no se imprime nada.



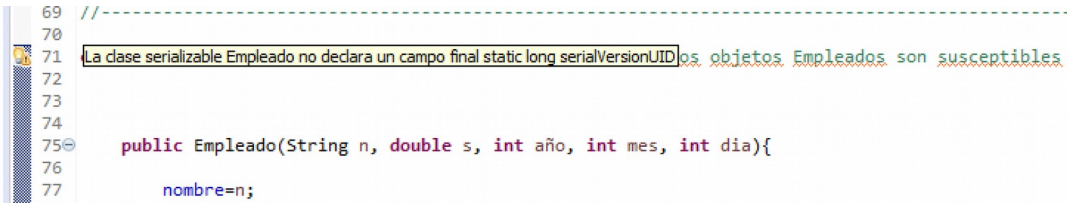
Sucede que el receptor tiene una versión antigua sin las actualizaciones y el programa no es capaz de leer el objeto generado con la nueva actualización.

¿Cuál sería la solución?

La solución pasa por declarar en nuestro programa nuestro propio "SerialVersionUID", no permitir que lo genere el compilador java.

Eso lo hacemos creando una constante de tipo long , de clase estática encapsulada que debe llamarse "serialVersionUID" y darle un valor, el que queramos, cuando cambiamos de versión, cambiamos ese valor y así sucesivamente.

Si el programa del emisor y el programa del receptor tiene declarado en cualquier punto del código esta constante de clase, da igual que el programa del emisor cambie, se actualice y no se haya informado de esa actualización al receptor, porque la huella no es generada automáticamente, y no cambia al modificar algo en el programa, porque la huella sigue siendo la misma para ambas partes.



Comprobamos que en nuestro programa tenemos unos avisos en unas clases, comprobamos que ese aviso nos dice que no tenemos declarada el campo "serialVersionUID".

El compilador nos lanza esta advertencia con el objetivo de que sabe que estamos creando una clase que es capaz de serializar objetos y el compilador advierte de que cree la constante serialVersionUID por si en el futuro cambia la versión de la aplicación.

```
70
71 class Empleado implements Serializable {// indicamos que los objetos Empleado
72
73     private static final long serialVersionUID = 1L;
74     public Empleado(String n, double s, int año, int mes, int dia){
75
```

```
123 //---------------------------------------------------------------------------
124
125 class Administrador extends Empleado{
126
127
128     private static final long serialVersionUID = 1L;
129
130
131     public Administrador(String n, double s, int año, int mes, int dia){
132
```

Nota: Repetir el proceso anterior y comprobar que ahora si lee el objeto cuando se ha cambiado la versión.

Ahora suponer el caso de que el emisor tiene la versión 1 y el receptor tiene la versión 2 , modificando el serialVersionUID en la clase Empleado y Administrador cuando estemos en la posición del receptor; es decir, borramos el fichero para dejar vacío el directorio y generamos el fichero con la versión 1 (caso del emisor). Ahora comentamos las tres líneas que me escribe el fichero y cambiamos la versión en las dos clases indicadas (supuesto del receptor) ejecutamos el programa y comprobamos no se lee el fichero generado con la versión 1.