

## Estructuras de Datos y Algoritmos I

### Grado en Ingeniería Informática, Curso 2º

## ACTIVIDAD 1

### Persistencia de Estructuras de Datos mediante Serialización.

#### Objetivos

- Aprender a almacenar y recuperar objetos en archivos de disco.
- Aprender a serializar objetos y estructuras de datos en disco.

#### Requerimientos

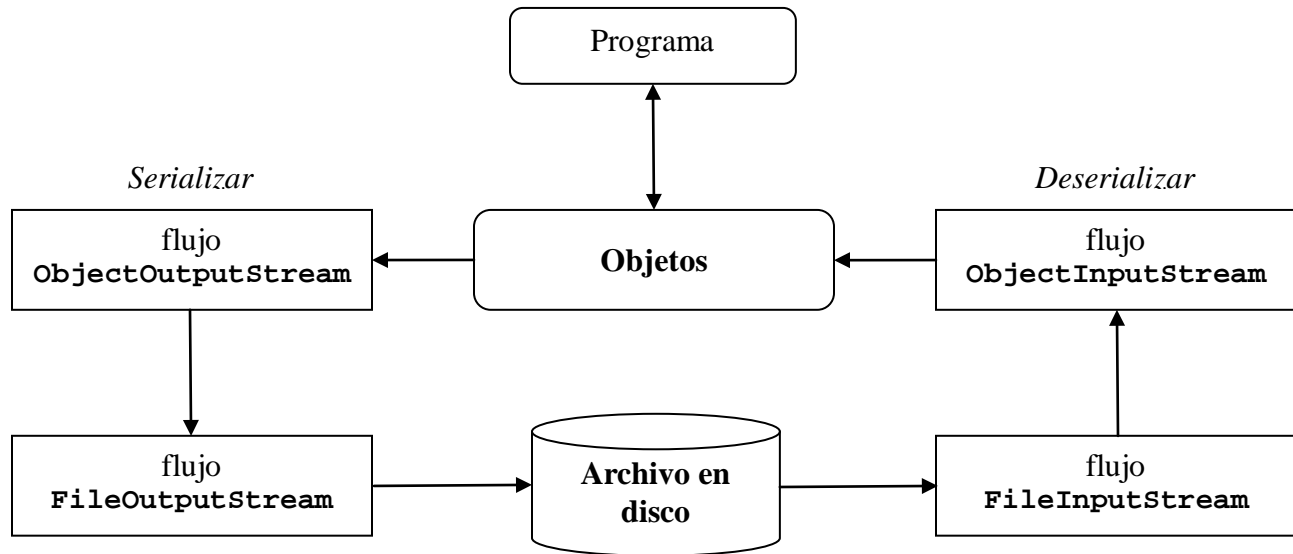
Para superar esta actividad se debe realizar lo siguiente:

- Dominar el proceso de serialización de objetos en disco.
- Dominar el proceso de serialización de estructuras de datos en disco.
- Pasar los respectivos test que se proporcionan.

#### Enunciado

Asumiendo que se conoce cómo escribir y leer grupos de datos a y desde un archivo (`FileWriter`, `FileReader`, `BufferedWriter`, `BufferedReader`, `FileInputStream`, `FileOutputStream`, `PrintWriter`, `PrintReader`, `DataInputStream`, `DataOutputStream`, etc.). Pero en un desarrollo orientado a objetos sabemos que debemos pensar en objetos; por lo tanto, a dicho grupo de datos no lo vamos a tratar de forma aislada; más bien se corresponderá con los atributos de un objeto, lo que nos conducirá a escribir y leer objetos a y desde un archivo.

Normalmente la operación de enviar una serie de objetos a un archivo en disco para hacerlos persistentes recibe el nombre de *serialización* o *seriación* (*serialization*). Y la operación de leer o recuperar su estado del archivo para reconstruirlos en memoria recibe el nombre de *deserialización* o *deseriación* (*deserialization*). Para realizar estas operaciones de forma automática, el paquete `java.io` proporciona las clases `ObjectOutputStream` y `ObjectInputStream`. Ambas clases dan lugar a flujos que procesan sus datos; en este caso, se trata de convertir el estado de un objeto (los atributos excepto las variables estáticas), incluyendo la clase del objeto y el prototipo de la misma, en una secuencia de bytes y viceversa. Por esta razón los flujos `ObjectOutputStream` y `ObjectInputStream` deben construirse sobre otros flujos que canalicen esos bytes a y desde el archivo. El esquema que ilustra todo este proceso se muestra en la figura siguiente.



Para poder serializar los objetos de una clase, ésta debe de implementar la interfaz **Serializable**. Se trata de una interfaz vacía; es decir, sin ningún método; y su propósito es simplemente identificar clases cuyos objetos se pueden serializar. Y cómo la interfaz **Serializable** está vacía no hay que escribir ningún método extra en la clase.

```
public class ClassName implements java.io.Serializable { // o ... Serializable
    // Code for the class
}
```

## Escribir objetos en un archivo

Un flujo de la clase **ObjectOutputStream** permite enviar datos de tipos primitivos y objetos hacia un flujo **OutputStream** o derivado. Concretamente, cuando se trate de almacenarlos en un archivo, utilizaremos in flujo **FileOutputStream**. Posteriormente, esos objetos podrán ser reconstruidos a través de un tipo **ObjectInputStream**.

Para escribir un objeto en un flujo **ObjectOutputStream** utilizaremos el método **writeObject**. Los objetos pueden incluir **String** y matrices, y el almacenamiento de los mismos puede combinarse con datos de tipos primitivos, ya que esta clase implementa la interfaz **DataOutput**. Este método lanzará la excepción **NotSerializableException** si se intenta escribir un objeto de una clase que no implementa la interfaz **Serializable**.

```
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("storeFile"));
oos.writeObject(anObject);
```

Por ejemplo, el siguiente código construye un objeto **ObjectOutputStream** sobre un **FileOutputStream**, y lo utiliza para almacenar un **String** y un objeto *Persona* en un archivo denominado *datos*.

```
FileOutputStream fos = new FileOutputStream("datos");
ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeUTF("Archivo datos");
oos.writeObject(new Persona(nombre, direccion, telefono));
oos.flush();
oos.close();
```

## Leer objetos desde un archivo

Un flujo de la clase **ObjectInputStream** permite recuperar datos de tipos primitivos y objetos desde un flujo **InputStream** o derivado. Concretamente, cuando se trate de datos de tipos primitivos y objetos almacenados en disco, utilizaremos un flujo **FileInputStream**. La clase **ObjectInputStream** implementa la interfaz **DataInput** para permitir leer también datos de tipos primitivos.

Para leer un objeto desde un flujo **ObjectInputStream** utilizaremos el método **readObject**. Si se almacenaron objetos y datos de tipos primitivos, debe ser recuperados en el mismo orden. Este método lanzará la excepción **ClassNotFoundException** si la definición de la clase para el objeto leído desde el flujo no está en el programa actual.

```
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("storeFile"));
ClassName recallObj = (ClassName)ois.readObject();
```

Por ejemplo, el siguiente fragmento de código construye un **ObjectInputStream** sobre un **FileInputStream**, y lo utiliza para recuperar un **String** y un objeto *Persona* de un archivo llamado *datos*.

```
FileInputStream fis = new FileInputStream("datos");
ObjectInputStream ois = new ObjectInputStream(fis);

String str = (String)ois.readUTF();
Persona persona = (Persona)ois.readObject();
ois.close();
```

## Serializar objetos que referencian a objetos

Cuando en un archivo se escribe un objeto que hace referencia a otros objetos, entonces todos los objetos accesibles desde el primero deben ser escritos en el mismo proceso para mantener así la relación existente entre todos ellos. Este proceso se lleva a cabo automáticamente por el método **writeObject**, que escribe el objeto especificado, recorriendo sus referencias a otros objetos recursivamente, escribiendo así todos ellos.

Análogamente, si el objeto recuperado del flujo por el método **readObject** hace referencia a otros objetos, **readObject** recorrerá sus referencias a otros objetos recursivamente, para recuperar todos ellos manteniendo la relación que existía entre ellos cuando fueron escritos.

## Ejemplo

// En el archivo *SerializableClass.java*

```
package org.eda1.actividad01.serializacion;

import java.io.Serializable;

public class SerializableClass implements Serializable {
    public int n;
    public String str;
    public Integer[] list = new Integer[4];

    public SerializableClass(int n, String str) {
        this.n = n;
        this.str = str;
        for (int i = 0; i < list.length; i++)
            list[i] = new Integer(i + 1);
    }
}
```

// En el archivo *ProgramSerialization.java*

```
package org.eda1.actividad01.serializacion;

import java.io.*;

public class ProgramSerialization {
    public static void main(String[] args) throws Exception {

        // objects used for serialization
        SerializableClass obj, recallObj;

        String directory = System.getProperty("user.dir");
        directory = directory + File.separator +
            "src" + File.separator +
            "org" + File.separator +
            "eda1" + File.separator +
            "actividad01" + File.separator +
            "serializacion" + File.separator;
        String storeFile = directory + "storeFile.dat";

        // object stream connected to file "storeFile" for output
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(storeFile));

        // initial object
        obj = new SerializableClass(45, "Data Structures and Algorithms class");
```

```
// output object info before copy to the file
System.out.println("Serialized object:");
System.out.println("    Integer: " + obj.n +
    " String: " + obj.str + " List: " +
    toString(obj.list));

// send object and close down the output stream
oos.writeObject(obj);
oos.flush();
oos.close();

// object stream connected to file "storeFile" for output
ObjectInputStream ois = new ObjectInputStream(new FileInputStream(storeFile));

// reconstruct object and allocate new current object
recallObj = (SerializableClass)ois.readObject();
ois.close();

// output object after recall from the file
System.out.println("Deserialized object:");
System.out.println("    Integer: " + recallObj.n +
    " String: " + recallObj.str + " List: " +
    toString(obj.list));
}

// returns a string that represents an array of objects
public static String toString(Object[] arr) {
    if (arr == null)
        return "null";
    else if (arr.length == 0)
        return "[]";

    // start with the left bracket
    String str = "[" + arr[0];

    // append all but the last element, separating items with a comma
    // polymorphism calls toString() for the array type
    for (int i = 1; i < arr.length; i++)
        str += ", " + arr[i];

    str += "];"

    return str;
}
}
```

## Personalizar la Serialización de objetos de colecciones (ED)

Para muchas clases la *serialización* por defecto de sus objetos funciona perfectamente. En otros casos, sin embargo, los programadores desean personalizar el proceso de escritura/lectura (**write/read**), cuando los datos de los objetos no se escriben y leen objetos de forma efectiva a y desde un archivo. Con objetos de colecciones, los elementos se generan dinámicamente y se almacenan siguiendo algún tipo de orden. El proceso de *deserialización* debe recuperar los elementos y entonces reconstruir la estructura de almacenamiento subyacente para la colección. Para una colección (ED) personalizada como las que veremos a continuación en los ejercicios (**LinkedList** y **ArrayList**) se debe de seguir el siguiente esquema:

```
public class CollectionClassName implements Interface<T>, Cloneable, Serializable {  
  
    // Code for the class  
  
    private void writeObject(ObjectOutputStream out) throws java.io.IOException {  
  
        // Write out internal serialization magic  
        out.defaultWriteObject();  
  
        // write out the elements of the collection (using out.writeObject(...))  
  
    }  
  
    private void readObject(ObjectInputStream in) throws IOException,  
                           ClassNotFoundException {  
  
        // Read in internal serialization magic  
        in.defaultReadObject();  
  
        // read the elements of the collection (using in.readObject())  
  
    }  
  
    // Code for the class  
}
```

## Ejercicio

Dadas las colecciones **LinkedList** y **ArrayList** (implementadas en el libro de Topp-Ford), implementar en cada caso todo lo necesario para realizar las operaciones de **writeObject** y **readObject**. Es decir, implementar las operaciones **writeObject** y **readObject**, siguiendo el esquema presentado para las colecciones, para que puedan serializar todos los datos gestionados por dichas estructuras de datos (**LinkedList** y **ArrayList**).

Además, implementar un programa que gestione y pueda **serializar** una estructura de datos que es un **ArrayList** de objetos *CiudadBarrios*, teniendo esta clase el nombre de la ciudad (**String**), su latitud (**Double**), su altitud (**Double**) y una **LinkedList** de **String** para almacenar los barrios más representativos de dicha ciudad. Los datos son los de las 8 provincias de Andalucía, y la salida del programa una vez deserializados los datos debe ser el siguiente:

```
[Almeria, 36.5, -2.28, {Regiones, Nueva_Andalucia, Cruz_Caravaca, Araceli,
Villablanca, Barrio_Alto, Paseo_Almeria, Zapillo}]
[Granada, 37.11, -3.35, {Albaicin, Sacromonte, Los Pajaritos, Chana, Cartuja,
Zaidin}]
[Malaga, 36.43, -4.25, {La_Goleta, El_Palo, La_Trinidad, La_Malagueta,
La_Merced}]
[Jaen, 37.46, -3.47, {La_Gloria, San_Ildefonso, Juderia, Las_Fuentezuelas}]
[Cordoba, 37.53, -4.47, {Del_Carmen, Arenal, San_Francisco, Fuensanta}]
[Sevilla, 37.23, -5.59, {Triana, Pineda, San_Pablo, Nervion, Macarena}]
[Cadiz, 36.32, -6.18, {San_Juan, Mentidero, Santa_Maria}]
[Huelva, 37.16, -6.57, {Reina_Victoria, Moret, Principe_Felipe}]
```

También se pide, que una vez serializada la estructura de datos en disco, modificar la estructura de datos en memoria, para que una vez deserializada se pueda comprobar que son diferentes.

Una aproximación parcial de la clase **CiudadBarrios** almacenada en el archivo *CiudadBarrios.java* podría ser la siguiente:

```
package org.eda1.actividad01.serializacionED;


import java.io.Serializable;


public class CiudadBarrios implements Serializable {
    public String ciudad;
    public Double latitud;
    public Double longitud;
    LinkedList<String> barrios = new LinkedList<String>();


    public CiudadBarrios(String ciudad, Double latitud, Double longitud) {
        this.ciudad = ciudad;
        this.latitud = latitud;
        this.longitud = longitud;
    }


    public boolean addBarrio(String barrio) {
        // Añadir el correspondiente código
    }

    public LinkedList<String> getBarrios() {
        // Añadir el correspondiente código
    }
}
```

 org.eda1.actividad01.serializacionED.ArrayList<T>
<ul style="list-style-type: none"> <li>serialVersionUID: long</li> <li>listSize: int</li> <li>listArr: T[]</li> <li>modCount: int</li> </ul>
<ul style="list-style-type: none"> <li>rangeCheck(index: int, msg: String, upperBound: int): void</li> <li>ArrayList(): void</li> <li>add(index: int, item: T): void</li> <li>remove(index: int): T</li> <li>add(item: T): boolean</li> <li>clear(): void</li> <li>contains(item: Object): boolean</li> <li>ensureCapacity(minCapacity: int): void</li> <li>get(index: int): T</li> <li>indexOf(item: Object): int</li> <li>isEmpty(): boolean</li> <li>iterator(): Iterator&lt;T&gt;</li> <li>listIterator(): ListIterator&lt;T&gt;</li> <li>listIterator(index: int): ListIterator&lt;T&gt;</li> <li>remove(item: Object): boolean</li> <li>set(index: int, item: T): T</li> <li>size(): int</li> <li>toString(): String</li> <li>trimToSize(): void</li> <li>toArray(): Object[]</li> <li>writeObject(out: ObjectOutputStream): void</li> <li>readObject(in: ObjectInputStream): void</li> <li>clone(): Object</li> </ul>

 org.eda1.actividad01.serializacionED.ArrayList<T>.IteratorImpl
<ul style="list-style-type: none"> <li>expectedModCount: int</li> <li>nextIndex: int</li> <li>prevIndex: int</li> </ul>
<ul style="list-style-type: none"> <li>IteratorImpl()</li> <li>hasNext(): boolean</li> <li>next(): T</li> <li>remove(): void</li> <li>checkIteratorState(): void</li> </ul>

 org.eda1.actividad01.serializacionED.ArrayList<T>.ListIteratorImpl
<ul style="list-style-type: none"> <li>ListIteratorImpl(index: int)</li> <li>add(item: T): void</li> <li>hasPrevious(): boolean</li> <li>nextIndex(): int</li> <li>previous(): T</li> <li>previousIndex(): int</li> <li>set(item: T): void</li> </ul>

 org.eda1.actividad01.serializacionED.CiudadBarrios
<ul style="list-style-type: none"> <li>ciudad: String</li> <li>latitud: Double</li> <li>longitud: Double</li> <li>barrios: LinkedList&lt;String&gt;</li> </ul>
<ul style="list-style-type: none"> <li>CiudadBarrios()</li> <li>CiudadBarrios(ciudad: String, latitud: Double, longitud: Double)</li> <li>setCiudad(ciudad: String): void</li> <li>setLatitud(latitud: Double): void</li> <li>setLongitud(longitud: Double): void</li> <li>addBarrio(barrio: String): boolean</li> <li>getBarrios(): LinkedList&lt;String&gt;</li> </ul>