Chapter 14 Serialization and File I/O

1. Introduction :

- Object have both **behaviour** and **state**. Behaviour lives within the class but state lives within each individual object.
- One does need to save the state of a game to play it from where he stopped or stop an video to start it from the same point. For all these we need to save the state of an object.
- Java provides many ways to save the state of an object.
    1. Serialization : Write the file that holds serialized objects. The programme can later read the objects from the file and inflate them back into living, breathing, heap inhabiting objects.
    2. Write a plain text file.
    3. Above two are not the only way off course. One can chose to write bytes instead of characters or any other primitive type etc.
- But regardless of method one use to save the state the fundamental I/O will be pretty much the same. Either one writes data to file disk or a stream coming from a network connection or Reading data through the same process in reverse i.e. reading either from file disk or a network conection.

2. Writing a serialized object to a file :

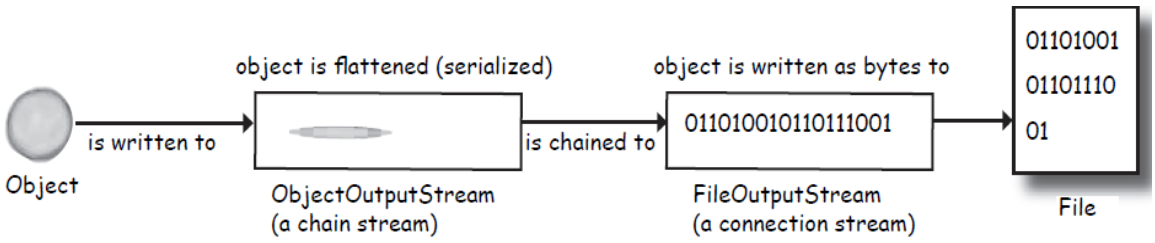
- Steps for serializing an object
    1. Make a FileOutputStream

       FileOutputStream fileStream = new FileOutputStream("MyGame.ser");
    2. Make an ObjectOutputStream

       ObjectOutputStream os = new ObjectOutputStream(fileStream);
    3. Write the Object

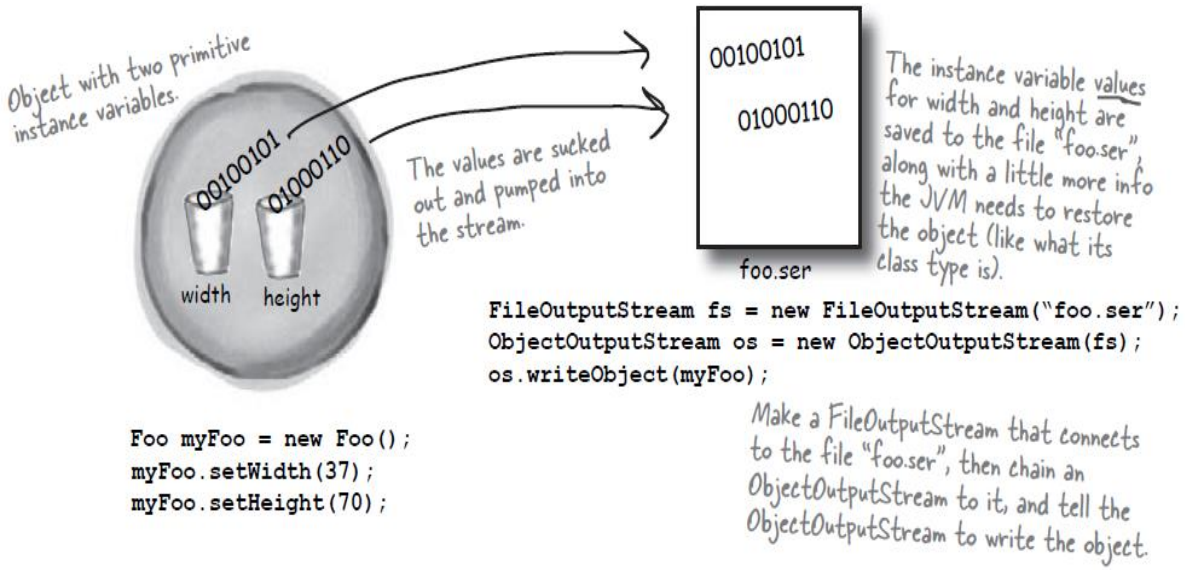
       os.writeObject(characterOne);
       os.writeObject(characterTwo);
       os.writeObject(characterThree);
    4. Close the ObjectOutputStream

       os.close();
- Always it will take two streams to hooked together to do something useful, this is called **Chaining**. Here we have ObjectOutputStream which turn object into data that can be written to a stream and FileOutputStream to write data to a file. Below diagram explains it correctly.

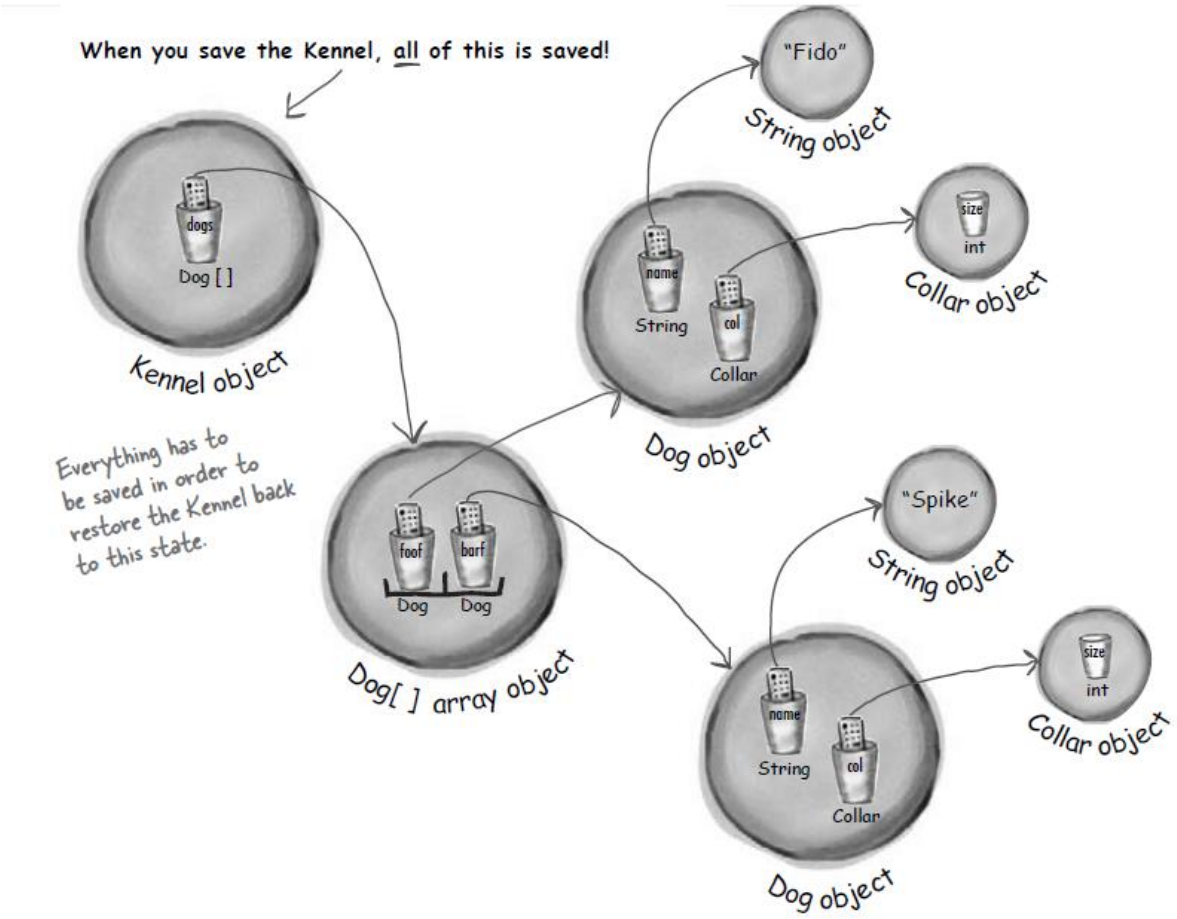


3. **<u>What happens when object is serialized</u>** :

- Object at heap have a state – the values of instance variables of object. These values make one instance of a class separate from the another instance of the same class.
- Serialized object saves the values of the instance variables so that an identical instance (object) can be brought back to life on the heap.
- Below diagram clearly explains it



- What exactly is an object's state ? that needs to be saved. Object can have a instance of a reference variable. That reference variable may inturn can have more reference variables. So during serialization all these objects i.e. object graph is serialized. Example if Kennel class have reference to Dog [] array object. The Dog[] array contains two Dog objects i.e. references of two Dog objects. Each Dog object hold references to a String and a Collar object. String is collection of characters and Collar contains int. When one tries to save Kennel i.e. Serialize Kennel then all of these will be saved. Below diagram explains it clearly.



4. **How to make a class Serializable** :
   - A class can be serializable if it implements **Serializable** interface. Serializable interface don't have any method, it is just a marker or tag interface which tells that the objects of that type are saveable through the serialization mechanism.
   - If any superclass of a class is Serializable then subclass is automatically serializable even if subclass doesn't explicitly declare '*implements Serializable'*.

5. **Class is not serialized whose reference is used**
   - Sometime it may happen that a class1 contains a reference variable for another class2(completely another class) and class2 don't implement serializable then during serialization compiler finds that class2 is not serializable hence throws 'NotSerializableException'. Example is shown below.

```
package inputOutputStream;

import java.io.*;
```

```
public class Pond implements Serializable
{
  private Duck duck = new Duck();
  public static void main (String[] args)
  {
        Pond myPond = new Pond();
        try
        {
              FileOutputStream fs = new FileOutputStream("Pond.ser");
              ObjectOutputStream os = new ObjectOutputStream(fs);
              os.writeObject(myPond);
              os.close();
        } catch(Exception ex)
              {
                    ex.printStackTrace();
              }
  }
}

class Duck
{
// duck code here
}
```

```
Output:
java.io.NotSerializableException: inputOutputStream.Duck
at inputOutputStream.Pond.main(Pond.java:14)
```

- To avoid such exception one should mark such field as '**transient**' keyword and the serialization will skip right over it. If we use transient keyword before duck reference as shown below then it will be skipped(from saving) during serialization and there will be no exception and programme executes correctly.

  ```
  private transient Duck duck = new Duck();
  ```

- Although most things in java class libraries are serializable still one cannot save many things like network connections, threads or file objects etc because they are dependent on a particular runtime experience and once the program shuts down there is no way to bring them back to life. They need to be created from scratch each time.
- If serializable is so useful then why by default every class implements serializable like if object class is made to implement serializable then it can be done. But it is no useful in many instances like one don't want to save a password object etc.
- If a class is not serializable i.e. don't implements Serializable then we can create a subclass of it and make it implements Serializable. This could be useful and then we can replace superclass instance with subclass. But when deserialization takes place then superclass constructor will run as new object is being created. (More on deserialization later).

- Similarly what happened to the variables which have been made transient during deserialization? The variable will be brought back as *null* regardless of the value it has during serialization. So what is the solution for such instances? There are two, first if the value is generic like dog has a collar but all collar objects are same then they can be reinitialize to the default value. Second is if value is contextual like this dog should have particular colour and design of collar then one should save the key values of the collar use them to recreate a brand new collar which is identical to original.
- What if two reference variables are holding the same object? Then it will be recognized in an Object graph that two objects are same and only one will be saved. During deserialization all references to that object are stored.

6. <u>Deserialization</u> :

- Deserialization is a lot like serialization in reverse.
- It will be done in following steps.

1. Make a FileInputStream

```
FileInputStream fileStream = new FileInputStream("MyGame.ser");
```

If file is not found then it will throw fileNotfoundException.

2. Make an ObjectInputStream

```
ObjectInputStream os = new ObjectInputStream(fileStream);
```

3. Read the objects

```
Object one = os.readObject();
Object two = os.readObject();
Object three = os.readObject();
```

Every time it calls readObject() then it will get the next object in the stream. So it will be read back in the same order they are written. Also if one tries to read more objects than what has been written then it will throw exception.

4. Cast the Objects

```
GameCharacter elf = (GameCharacter) one;
GameCharacter troll = (GameCharacter) two;
GameCharacter magician = (GameCharacter) three;
```

The return type of readObject() will be 'Object' hence they needed to be cast back to their original type.

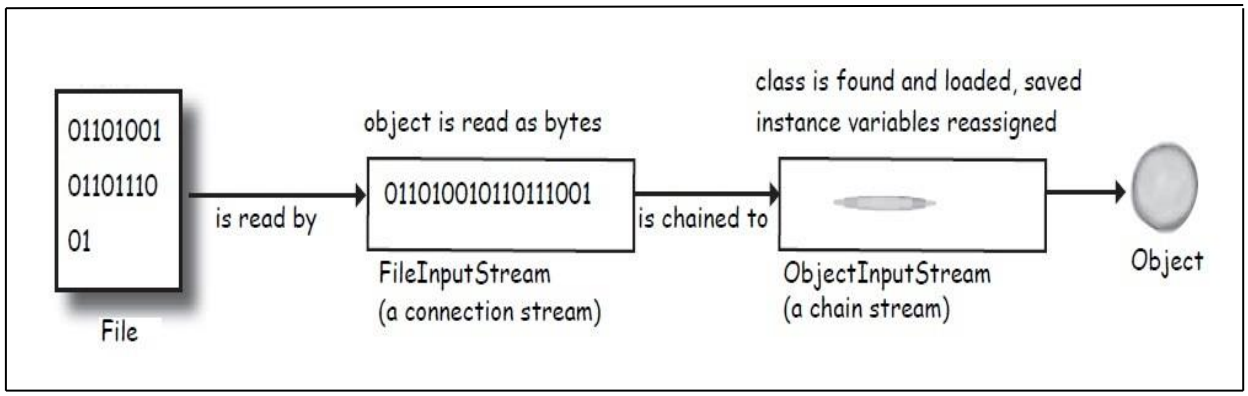
5. Close the ObjectInputStream

```
os.close();
```

Closing the stream at top closes the ones underneath i.e. FileInputStream(and file) will be closed automatically.

- **<u>What happens during deserialization</u>:**

When an Object is deserialized, JVM tries to bring back that object to life making a new object at the heap having the same state it had at the time of serialization. Except the transient variables which will be initialized as null (for object references) or default values(for primitives).



1. The object is **read** from the stream.

2. The JVM determines (through info stored with the serialized object) the object's **class type.**

3. The JVM attempts to **find and load** the object's **class**. If the JVM can't find and/or load the class, the JVM throws an exception and the deserialization fails.

4. A new object is given space on the heap, but the **serialized object's constructor does NOT run**! Obviously, if the constructor ran, it would restore the state of the object back to its original 'new' state, and that's not what we want. We want the object to be restored to the state it had *when it was serialized*, not when it was first created.

5. If the object has a non-serializable class somewhere up its inheritance tree, the **constructor for that non-serializable class will run** along with any constructors above that (even if they're serializable). Once the constructor chaining begins, you can't stop it, which means all superclasses, beginning with the first non-serializable one, will reinitialize their state.

6. The object's **instance variables are given the values from the serialized state**. Transient variables are given a value of null for object references and defaults (0, false, etc.) for primitives.

- Static variables are not encouraged for serialization as they dont belong to object but classes.

6. Example for both Serialization and Deserialization :

- Class GameCharacter :

```
package ch14SerializationAndFileIO;

import java.io.Serializable;

public class GameCharacter implements Serializable
{
    int power;
    String type;
    String[] weapons;

    public GameCharacter(int p, String t, String[] w)
    {
        power = p;
        type = t;
        weapons = w;
    }

    public int getPower() {
        return power;
    }

    public String getType() {
        return type;
    }

    public String getWeapons() {
        String weaponList = "";
        for (int i = 0; i < weapons.length; i++)
        {
            weaponList += weapons[i] + " ";
        }
        return weaponList;
    }
}
```

- Class GameSaverTest :

```
package ch14SerializationAndFileIO;

import java.io.*;

public class GameSaverTest
{
    public static void main (String[] args) {
        GameCharacter one = new GameCharacter(50, "Elf", new String[] {"bow",
"sword", "dust"});
```

```
        GameCharacter two = new GameCharacter(200, "Troll", new String[]
{"bare hands", "big axe"});
        GameCharacter three = new GameCharacter(120, "Magician", new String[]
{"spells", "invisibility"});

        try {
            ObjectOutputStream os = new ObjectOutputStream(new
FileOutputStream("D:/Game.txt"));
            os.writeObject(one);
            os.writeObject(two);
            os.writeObject(three);
            os.close();
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }

        one = null;
        two = null;
        three = null;

        try {
            ObjectInputStream is = new ObjectInputStream(new
FileInputStream("D:/Game.txt"));
            GameCharacter oneRestore = (GameCharacter) is.readObject();
            GameCharacter twoRestore = (GameCharacter) is.readObject();
            GameCharacter threeRestore = (GameCharacter) is.readObject();

            System.out.println("One's type: " + oneRestore.getType());
            System.out.println("Two's type: " + twoRestore.getType());
            System.out.println("Three's type: " + threeRestore.getType());
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
Output:
One's type: Elf
Two's type: Troll
Three's type: Magician
```

Note : If we mark 'type' as transient i.e.

```
transient String type;
```

in class GameCharacter.java then output would be

```
One's type: null
Two's type: null
Three's type: null
```

BULLET POINTS

- You can save an object's state by serializing the object.
- To serialize an object, you need an ObjectOutputStream (from the java.io package)
- Streams are either connection streams or chain streams
- Connection streams can represent a connection to a source or destination, typically a file, network socket connection, or the console.
- Chain streams cannot connect to a source or destination and must be chained to a connection (or other) stream.
- To serialize an object to a file, make a FileOuputStream and chain it into an ObjectOutputStream.
- To serialize an object, call *writeObject(theObject)* on the ObjectOutputStream. You do not need to call methods on the FileOutputStream.
- To be serialized, an object must implement the Serializable interface. If a superclass of the class implements Serializable, the subclass will automatically be serializable even if it does not specifically declare *implements Serializable*.
- When an object is serialized, its entire object graph is serialized. That means any objects referenced by the serialized object's instance variables are serialized, and any objects referenced by those objects...and so on.
- If any object in the graph is not serializable, an exception will be thrown at runtime, unless the instance variable referring to the object is skipped.
- Mark an instance variable with the *transient* keyword if you want serialization to skip that variable. The variable will be restored as null (for object references) or default values (for primitives).
- During deserialization, the class of all objects in the graph must be available to the JVM.
- You read objects in (using readObject()) in the order in which they were originally written.
- The return type of readObject() is type Object, so deserialized objects must be cast to their real type.
- Static variables are not serialized! It doesn't make sense to save a static variable value as part of a specific object's state, since all objects of that type share only a single value—the one in the class.