JAVA BEANS

A JavaBean is just a standard

1. All properties private (use getters/setters)
2. A public no-argument constructor
3. Implements Serializable.

Serializable Objects

To *serialize* an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. A Java object is *serializable* if its class or any of its superclasses implements either the java.io.Serializable interface or its subinterface, java.io.Externalizable. *Deserialization* is the process of converting the serialized form of an object back into a copy of the object.

For example, the java.awt.Button class implements the Serializable interface, so you can serialize a java.awt.Button object and store that serialized state in a file. Later, you can read back the serialized state and deserialize into a java.awt.Button object.

The Java platform specifies a default way by which serializable objects are serialized. A (Java) class can override this default serialization and define its own way of serializing objects of that class. The Object Serialization Specification describes object serialization in detail.

When an object is serialized, information that identifies its class is recorded in the serialized stream. However, the class's definition ("class file") itself is not recorded. It is the responsibility of the system that is deserializing the object to determine how to locate and load the necessary class files. For example, a Java application might include in its classpath a JAR file that contains the class files of the serialized object(s) or load the class definitions by using information stored in the directory, as explained later in this lesson.

Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable. Serializable objects can be written to streams, and hence files, object databases, anything really.

Also, there is no syntactic difference between a JavaBean and another class -- a class is a JavaBean if it follows the standards.

There is a term for it because the standard allows libraries to programmatically do things with class instances you define in a predefined way. For example, if a library wants to stream any object you pass into it, it knows it can because your object is serializable (assuming the lib requires your objects be proper JavaBeans).

https://www.tutorialspoint.com/java/java_serialization.htm

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out −

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object −

```
public final Object readObject() throws IOException, ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

Notice that for a class to be serialized successfully, two conditions must be met −

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

Serializing an Object

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note − When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

Example

```
import java.io.*;
public class SerializeDemo {

   public static void main(String [] args) {
      Employee e = new Employee();
      e.name = "Reyan Ali";
      e.address = "Phokka Kuan, Ambehta Peer";
      e.SSN = 11122333;
      e.number = 101;

      try {
         FileOutputStream fileOut =
         new FileOutputStream("/tmp/employee.ser");
         ObjectOutputStream out = new ObjectOutputStream(fileOut);
         out.writeObject(e);
         out.close();
         fileOut.close();
         System.out.printf("Serialized data is saved in /tmp/employee.ser");
      } catch (IOException i) {
         i.printStackTrace();
      }
   }
}
```

Deserializing an Object

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output −

Example

```
import java.io.*;
public class DeserializeDemo {

   public static void main(String [] args) {
      Employee e = null;
      try {
         FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         e = (Employee) in.readObject();
         in.close();
         fileIn.close();
      } catch (IOException i) {
         i.printStackTrace();
         return;
      } catch (ClassNotFoundException c) {
         System.out.println("Employee class not found");
         c.printStackTrace();
         return;
      }

      System.out.println("Deserialized Employee...");
      System.out.println("Name: " + e.name);
      System.out.println("Address: " + e.address);
      System.out.println("SSN: " + e.SSN);
      System.out.println("Number: " + e.number);
   }
}
```

This will produce the following result −

Output

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

Here are following important points to be noted −

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

For further understanding:

https://www.javatpoint.com/serialization-in-java

A JavaBean is a Java class that should follow the following conventions:

- It should have a no-arg constructor.
- It should be Serializable.
- It should provide methods to set and get the values of the properties, known as getter and setter methods.
- it is a reusable software component.
- A bean encapsulates many objects into one object so that we can access this object from multiple places.
- Moreover, it provides easy maintenance.

A JavaBean property is a named feature that can be accessed by the user of the object. The feature can be of any Java data type, containing the classes that you define.

A JavaBean property may be read, write, read-only, or write-only. JavaBean features are accessed through two methods in the JavaBean's implementation class:

1. getPropertyName ()

For example, if the property name is firstName, the method name would be getFirstName() to read that property. This method is called the accessor.

2. setPropertyName ()

For example, if the property name is firstName, the method name would be setFirstName() to write that property. This method is called the mutator.

JavaBeans Example

Consider a student class with few properties −

```
package com.tutorialspoint;

public class StudentsBean implements java.io.Serializable {
   private String firstName = null;
   private String lastName = null;
```

```
  private int age = 0;

  public StudentsBean() {
  }
  public String getFirstName(){
    return firstName;
  }
  public String getLastName(){
    return lastName;
  }
  public int getAge(){
    return age;
  }
  public void setFirstName(String firstName){
    this.firstName = firstName;
  }
  public void setLastName(String lastName){
    this.lastName = lastName;
  }
  public void setAge(Integer age){
    this.age = age;
  }
}
```

Accessing JavaBeans

The **useBean** action declares a JavaBean for use in a JSP. Once declared, the bean becomes a scripting variable that can be accessed by both scripting elements and other custom tags used in the JSP. The full syntax for the useBean tag is as follows −

<jsp:useBean id = "bean's name" scope = "bean's scope" typeSpec/>

Here values for the scope attribute can be a **page, request, session** or **application based** on your requirement. The value of the **id** attribute may be any value as a long as it is a unique name among other **useBean declarations** in the same JSP.

Following example shows how to use the useBean action −

```
<html>
  <head>
    <title>useBean Example</title>
  </head>

  <body>
    <jsp:useBean id = "date" class = "java.util.Date" />
    <p>The date/time is <%= date %>
  </body>
</html>
```

You will receive the following result − −

```
The date/time is Thu Sep 30 11:18:11 GST 2010
```

Accessing JavaBeans Properties

Along with **<jsp:useBean...>** action, you can use the **<jsp:getProperty/>** action to access the get methods and the **<jsp:setProperty/>** action to access the set methods. Here is the full syntax −

```
<jsp:useBean id = "id" class = "bean's class" scope = "bean's scope">
  <jsp:setProperty name = "bean's id" property = "property name"
    value = "value"/>
  <jsp:getProperty name = "bean's id" property = "property name"/>
  ...........
</jsp:useBean>
```

The name attribute references the id of a JavaBean previously introduced to the JSP by the useBean action. The property attribute is the name of the **get** or the **set** methods that should be invoked.

Following example shows how to access the data using the above syntax −

```
<html>
  <head>
    <title>get and set properties Example</title>
  </head>

  <body>
    <jsp:useBean id = "students" class = "com.tutorialspoint.StudentsBean">
      <jsp:setProperty name = "students" property = "firstName" value = "Zara"/>
      <jsp:setProperty name = "students" property = "lastName" value = "Ali"/>
      <jsp:setProperty name = "students" property = "age" value = "10"/>
    </jsp:useBean>

    <p>Student First Name:
      <jsp:getProperty name = "students" property = "firstName"/>
    </p>

    <p>Student Last Name:
      <jsp:getProperty name = "students" property = "lastName"/>
    </p>

    <p>Student Age:
      <jsp:getProperty name = "students" property = "age"/>
    </p>

  </body>
```

```
</html>
```

Let us make the **StudentsBean.class** available in CLASSPATH. Access the above JSP. the following result will be displayed −

Student First Name: Zara

Student Last Name: Ali

Student Age: 10

Advantages of JavaBean

- The JavaBean properties and methods can be exposed to another application.
- It provides an easiness to reuse the software components.

Disadvantages of JavaBean

- JavaBeans are mutable. So, it can't take advantages of immutable objects.
- Creating the setter and getter method for each property separately may lead to the boilerplate code.