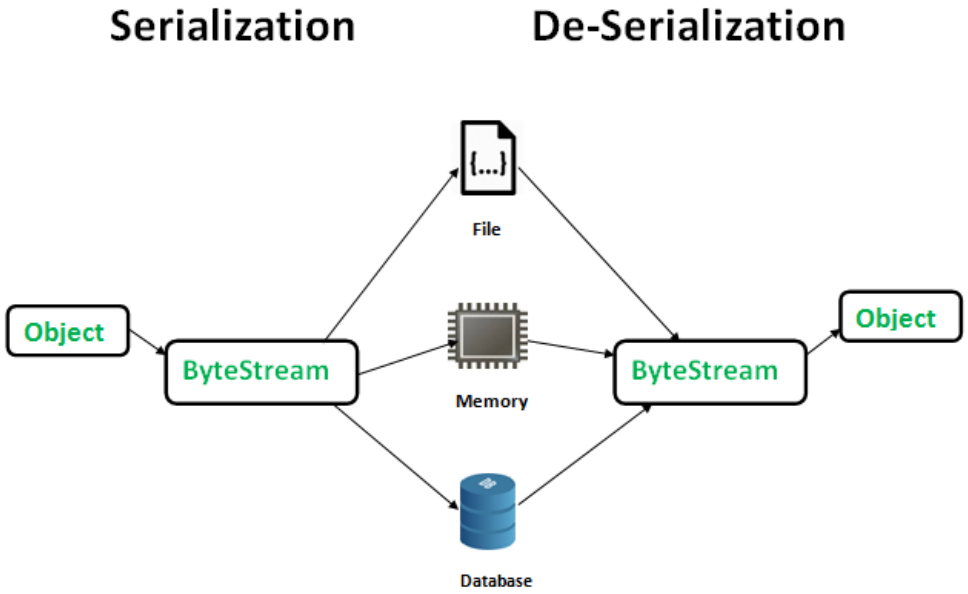
Serialization and Deserialization in Java

Serialization and Deserialization

Serialization is a mechanism of converting the state of an object into a byte stream.
Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



Serialization and Deserialization

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform

To make a Java object serializable we implement
the **java.io.Serializable** interface.
The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

The ObjectInputStream class contains **readObject()** method for deserializing an object.

Advantages of Serialization
1. To save/persist state of an object.
2. To travel an object across a network.

Serialization and Deserialization

Points to remember

1. If a parent class has implemented Serializable interface, then child class doesn't need to implement it but vice-versa is not true.

2. Only non-static data members are saved via Serialization process.

3. Static data members and transient data members are not saved via Serialization process. So, if you don't want to save value of a non-static data member then make it transient.

4. Constructor of object is never called when an object is deserialized.

EXAMPLE

Create class named Example:

```java
import java.io.Serializable;

public class Example implements Serializable {

    transient int transientInt;
    static int staticInt;
    String name;
    int age;

    public Example(String name, int age, int transientInt, int staticInt) {
        this.name = name;
        this.age = age;
        this.transientInt = transientInt;
        this.staticInt = staticInt;
    }

}
```

EXAMPLE

Create class SerialExample

```
public class SerialExample {

    public static void printData(Example object1) {
        System.out.println("name = " + object1.name);
        System.out.println("age = " + object1.age);
        System.out.println("transientInt = " + object1.transientInt);
        System.out.println("staticInt = " + object1.staticInt);
    }

    public static void main(String[] args) {
        Example object = new Example( name: "Name", age: 20, transientInt: 2, staticInt: 1000);
        String filename = "test.txt";

        // Serialization
        try {

            // Saving of object in a file
            FileOutputStream file = new FileOutputStream
                (filename);
            ObjectOutputStream out = new ObjectOutputStream
                (file);

            // Method for serialization of object
            out.writeObject(object);

            out.close();
            file.close();

            System.out.println("Object has been serialized\n"
                + "Data before Deserialization.");
            printData(object);

            // value of static variable changed
            object.staticInt = 2000;
        } catch (IOException ex) {
            System.out.println("IOException is caught");
        }

        // Deserialization
```

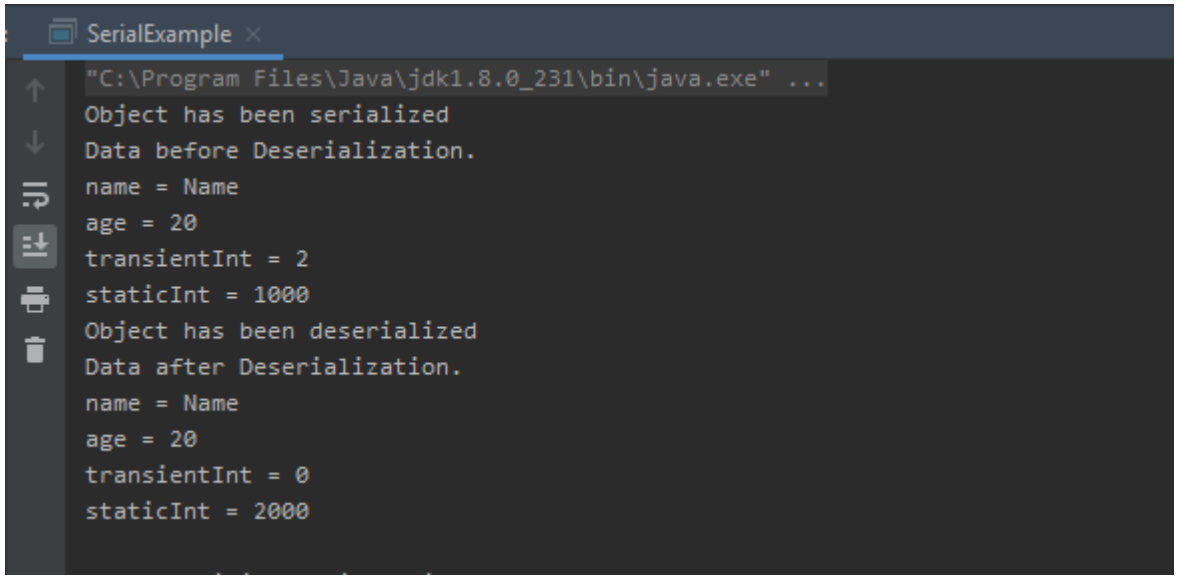
EXAMPLE

```
        // Deserialization
        try {

            // Reading the object from a file
            FileInputStream file = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(file);

            // Method for deserialization of object
            object = (Example) in.readObject();

            in.close();
            file.close();
            System.out.println("Object has been deserialized\n"
                + "Data after Deserialization.");
            printData(object);

        } catch (IOException ex) {
            System.out.println("IOException is caught");
        } catch (ClassNotFoundException ex) {
            System.out.println("ClassNotFoundException" +
                " is caught");
        }
    }
}
```

EXAMPLE

Observe the output

The value of transientInt and staticInt has changed after the deserialization.Why?



EXAMPLE

The reason being transientInt was marked as transient and staticInt was static.
In case of **transient variables:-** A variable defined with transient keyword is not serialized during serialization process. This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0).
In case of **static Variables:-** A variable defined with static keyword is not serialized during serialization process. This variable will be loaded with current value defined in the class during deserialization.

ASSIGNMENT

Create a class Student with following attributes:

String name;

String phoneNum;

int indexNum;

transient int recordNum;

Create default constructor without parameters

Create constructor which will assign values to all attributes

ASSIGNMENT

Create a class StudentFileStream with 3 methods:

Public void printData(Student student) which will print the values of Student object attributes

Public void writeTofile(Student student, String filename) which will write the student object into given file and print the attribute values after the serialization

Public Student readFromFile(String filename) which will read from file print the attribute values and return the deserialized student

Create class Demo with main method

Create Student object, write the student object in a file using created method and then read it from the file.

Example 2

1- <SerializationDef.java>.
2- <SerializationLib.java>.
3- <SerializationDemo.java>.

- **SerializationDef.java.**

- This file defines the Java class that we'll use for serialization. This class represents a simple Java bean carrying some properties and getter/setter methods. By default, all the properties get serialized.
- The ***<transient>*** is a keyword in Java. It marks a field to exclude from serialization. You can use this keyword for a variable which you don't want to be the part of the persistent state of an object.
- **A static member of the class does not get serialized.** A static member is associated with the class, not with the object of the class. It gets memory once during the loading of the class.

Example 2

```
import java.io.Serializable;

public class SerializationDef implements Serializable {

    private String Product;
    private String Feature;
    transient private int FeatureCount;

    @Override
    public String toString(){
        return "Summary[Product("+Product+"),Feature("+Feature+"),"
                + "FeatureCount("+FeatureCount+")]";
    }

    public String getProduct() {
        return Product;
    }

    public void setProduct(String product) {
        this.Product = product;
    }

    public String getFeature() {
        return Feature;
    }

    public void setFeature(String feature) {
        this.Feature = feature;
    }

    public int getFeatureCount() {
        return FeatureCount;
    }

    public void setFeatureCount(int count) {
        this.FeatureCount = count;
    }
}
```

Example 2

1- <SerializationDef.java>.
2- <SerializationLib.java>.
3- <SerializationDemo.java>.

- **SerializationLib.java.**

- For serialization, you'll need to copy the object to a file. You may then seek to deserialize it from the same file. For all this, you'll need helper functions, you can find them in the below code snippet.
- You'll notice from the code below that we are using the <***ObjectOutputStream***> and <***ObjectInputStream***> classes for serialization.
- Their methods take the <***Object***> class variable as the argument which is the parent class of all the classes in Java.

Example 2

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

public class SerializationLib {

    // Do serialize the Java object and save it to a file
    public static void doSerialize(Object obj, String outputFile)
            throws IOException {
        FileOutputStream fileTowrite = new FileOutputStream(outputFile);
        ObjectOutputStream objTowrite = new ObjectOutputStream(fileTowrite);
        objTowrite.writeObject(obj);

        fileTowrite.close();
    }

    // Do deserialize the Java object from a given file
    public static Object doDeserialize(String inputFile) throws IOException,
            ClassNotFoundException {
        FileInputStream fileToread = new FileInputStream(inputFile);
        ObjectInputStream objToread = new ObjectInputStream(fileToread);
        Object obj = objToread.readObject();
        objToread.close();
        return obj;
    }
}
```

Example 2

1- <SerializationDef.java>.
2- <SerializationLib.java>.
3- <SerializationDemo.java>.

- **SerializationDemo.java.**
- This is main file to demonstrate the serialization process.

Example 2

```
import java.io.IOException;

public class SerializationDemo {

    public static void main(String[] args) {

        String outputFile="serializationdemo.txt";
        SerializationDef def = new SerializationDef();
        def.setProduct("testProduct");
        def.setFeature("testFeature");
        def.setFeatureCount(10);

        // Serialize the object into a file.
        try {
            SerializationLib.doSerialize(def, outputFile);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        // Deserialize from a file into an object.
        SerializationDef defNext = null;
        try {
            defNext = (SerializationDef) SerializationLib.doDeserialize(outputFile);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }

        System.out.println("def():\n --"+"\n  |\n  "+def);
        System.out.println(System.lineSeparator());
        System.out.println("defNext():\n --"+"\n  |\n  "+defNext);
    }

}
```

Example 3- Java serialization with Inheritance

- There occur two cases when we use serialization with inheritance.
- **1-** When the parent class implements the Serializable interface, the child class does it automatically.
  2- If parent class doesn't implement the Serializable interface, then its state won't transform into a byte stream while serializing the child class instance.
- For managing the 2nd case, you need to implement the following two methods in the Child class.
- **1- <*readObject()*>.**
  2- <*writeObject()*>.

Example 3- Java serialization with Inheritance

- *<ParentClass.java>*
- The first part of the project is the *<ParentClass.java>* file which doesn't implement the Serializable interface.

```
public class ParentClass {

    private String Product;
    private int ProductId;

    public String getProduct() {
        return Product;
    }

    public void setProduct(String product) {
        this.Product = product;
    }

    public int getProductId() {
        return ProductId;
    }

    public void setProductId(int Id) {
        this.ProductId = Id;
    }
}
```

Example 3- Java serialization with Inheritance

- ***<ChildClass.java>***
- the ***<ChildClass.java>*** file which defines the read/write object method for preparing the stream of parent class state.
- The order of reading/writing data to the byte stream would remain the same.
- The child class is implementing the ***<ObjectInputValidation>*** interface. It'll allow overriding of some methods where you can add some business logic to ensure the data integrity.

Example 3- Java serialization with Inheritance

```java
import java.io.IOException;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectInputValidation;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class ChildClass extends ParentClass implements Serializable,
        ObjectInputValidation {

    private String Brand;

    public String getBrand() {
        return Brand;
    }

    public void setBrand(String brand) {
        this.Brand = brand;
    }

    @Override
    public String toString() {
        return "Summary[ ProductId=" + getProductId() + ", Product=" + getProduct()
                + ", Brand=" + getBrand() + " ]";
    }

    // adding helper method for serialization to save/initialize parent class
    // state
    private void readObject(ObjectInputStream ois)
            throws ClassNotFoundException, IOException {
        ois.defaultReadObject();

        // notice the order of read and write should be same
        setProductId(ois.readInt());
        setProduct((String) ois.readObject());

    }
```

```java
    private void writeObject(ObjectOutputStream oos) throws IOException {
        oos.defaultWriteObject();

        oos.writeInt(getProductId());
        oos.writeObject(getProduct());
    }

    @Override
    public void validateObject() throws InvalidObjectException {
        // validate the object here
        if (Brand == null || "".equals(Brand))
            throw new InvalidObjectException("Brand is not set or empty.");
        if (getProductId() <= 0)
            throw new InvalidObjectException("ProductId is not set or zero.");
    }
}
```

Example 3- Java serialization with Inheritance

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

public class SerializationLib {

    // Do serialize the Java object and save it to a file
    public static void doSerialize(Object obj, String outputFile)
            throws IOException {
        FileOutputStream fileTowrite = new FileOutputStream(outputFile);
        ObjectOutputStream objTowrite = new ObjectOutputStream(fileTowrite);
        objTowrite.writeObject(obj);

        fileTowrite.close();
    }

    // Do deserialize the Java object from a given file
    public static Object doDeserialize(String inputFile) throws IOException,
            ClassNotFoundException {
        FileInputStream fileToread = new FileInputStream(inputFile);
        ObjectInputStream objToread = new ObjectInputStream(fileToread);
        Object obj = objToread.readObject();
        objToread.close();
        return obj;
    }
}
```

Example 3- Java serialization with Inheritance

- *<InheritanceDemo.java>*
- The demo class file is used to serialize/deserialize the child class data including the parent. And see whether we can retrieve the parent class state from the serialized form.

```
import java.io.IOException;

public class InheritanceDemo {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        String fileName = "childclass.txt";
        ChildClass childClass = new ChildClass();
        childClass.setProductId(21);
        childClass.setProduct("Blog");
        childClass.setBrand("TechBeamers");

        try {
            SerializationLib.doSerialize(childClass, fileName);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }

        try {
            ChildClass newChild = (ChildClass) SerializationLib
                    .doDeserialize(fileName);
            System.out.println("ChildClass output:  \n  |\n   --" + newChild);
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }
}
```

Manage class refactoring with Java serialization

- Java serialization permits to refactor the underlying class. Following is the list of changes that are allowed in a class and won't disturb the deserialization process.
- You can add new members to the class.
- Switching of a variable from transient to non-transient is allowed. But Serialization will consider such variables as new.
- Make a variable from static to non-static. Serialization will count it as a new variable.
- However, Java imposes a condition for all these changes to work. You can fulfill it by adding a unique identifier, the ***<serialVersionUID>*** in the class to track the modifications under a common tag. By default, serialization will automatically compute the ***<serialVersionUID>*** by going through all the fields and methods. That's why if you try to alter any class variable without manually specifying the version identifier, the JVM will throw the ***<java.io.InvalidClassExceptio>*** as it detects a change in the identifier value.

How to generate a <serialVersionUID>?

command line utility called the ***<serialver>***.

```
C:\Users\Stojan\eclipse-workspace\Serialization-4\src>javac InheritanceDemo.java

C:\Users\Stojan\eclipse-workspace\Serialization-4\src>java InheritanceDemo
ChildClass output:
  |
   --Summary[ ProductId=21, Product=Blog, Brand=TechBeamers ]

C:\Users\Stojan\eclipse-workspace\Serialization-4\src>serialver InheritanceDemo
Class InheritanceDemo is not Serializable.

C:\Users\Stojan\eclipse-workspace\Serialization-4\src>serialver ChildClass
ChildClass:    private static final long serialVersionUID = 1363776251222308667L;
```

- Add default serial version ID, or
- Add generated serial version ID.

2.3.1.3- Assign the value of your choice.

Just pick a number you like and set as a serial version ID. But do postfix the number with an “L”.

```
private static final long serialVersionUID = 21L;
```

Java Externalizable Interface for Serialization

- The default serialization method is not secure and has performance issues.
- Performance issues with default serialization.
  - **1-** Serialization depends on the recursion mechanism. When serialization of a child class object starts, it triggers the serialization of other instance variables from the chain of parent classes which continues until it reaches the Object class of these variables. It leads to a lot of overheads.
  - **2-** While serializing an object class description information is attached to the stream. Lots of data and metadata stretch down the performance.
  - **3-** Serialization also needs a serial version ID for tracking class level changes. If you don't set it manually, serialization calculates it automatically by going through all the fields and methods. More the size of the class more will be the time to calculate the value. So this is again a potential performance issue.
  - **4-** We can solve all of the above issues with Externalization interface.

Externalizable interface

- Externalization is an extension to the Serializable interface.
- If you want to externalize an object, then your class needs to implement the Externalizable interface and a default public constructor.
- In the Externalization process, only the identity of the class is added to the serialization stream. And the class is responsible for the storage and retrieval of the instance information. It gives complete control of what to add and what to leave during serialization. The basic serialization doesn't offer the similar flexibility.
- Externalizable is not a marker interface instead it exposes two methods – writeExternal and readExternal.
- The target class implements these methods to seize full control over the format and contents of the stream relating to the object and its supertypes.
- These methods must have the ability to coordinate with the object's supertypes to save its state. They supersede the tailor-made implementation of ***<writeObject>*** and ***<readObject>*** methods.

Externalizable interface - Example

- **UIMap.java** this file defines a class which implements the Externalizable interface and provides the *<readExternal()>* and *<writeExternal()>* methods.

```java
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class UIMap implements Externalizable {

    private int id;
    private String locator;
    private String value;

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(id);
        out.writeObject(locator + "$$");
        out.writeObject("##" + value);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
        id = in.readInt();
        // Retrieve data in the same sequence as written
        locator = (String) in.readObject();
        if (!locator.endsWith("$$"))
            throw new IOException("data integrity failed.");
        locator = locator.substring(0, locator.length() - 2);
        value = (String) in.readObject();
        if (!value.startsWith("##"))
            throw new IOException("data integrity failed.");
        value = value.substring(2);
    }
```

```java
    @Override
    public String toString() {
        return "UIMap[ id=" + id + ",locator=" + locator + ",value=" + value + " ]";
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getLocator() {
        return locator;
    }

    public void setLocator(String locator) {
        this.locator = locator;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```

Externalizable interface - Example

- **UIMapDemo.java**, this file will create the UIMap object and test the Java serialization process via the Externalizable interface.

```java
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class UIMap implements Externalizable {

    private int id;
    private String locator;
    private String value;

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(id);
        out.writeObject(locator + "$$");
        out.writeObject("##" + value);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException,
            ClassNotFoundException {
        id = in.readInt();
        // Retrieve data in the same sequence as written
        locator = (String) in.readObject();
        if (!locator.endsWith("$$"))
            throw new IOException("data integrity failed.");
        locator = locator.substring(0, locator.length() - 2);
        value = (String) in.readObject();
        if (!value.startsWith("##"))
            throw new IOException("data integrity failed.");
        value = value.substring(2);
    }

    @Override
    public String toString() {
        return "UIMap[ id=" + id + ",locator=" + locator
                ",value=" + value + " ]";
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getLocator() {
        return locator;
    }

    public void setLocator(String locator) {
        this.locator = locator;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }
}
```