**IO-Streams**

**What is a stream?**

|  |
| --- |
| 1. A stream is a flow of data 2. A stream is a pipe through which we can read or write the data. 3. In java we have to use streams to read or write the data from source to destination.   **Streams are of 2 types**   1. Byte streams 2. Character streams |

**Byte Streams**

|  |
| --- |
| These streams are used to read or write the data byte by byte.  Byte streams are of 2 types   1. InputStreams 2. OutputStreams   **Note: By using a byte stream we can read or write a single character at a time which can be fit in 1 byte memory** |

Example to read the data from a file using a byte stream called FileInputStream.

Abbo.txt

|  |
| --- |
| hello hi how r u  namasthey nenu mee madhu |

ByStreamDemo1.java

import java.io.FileInputStream;

/\*\*

 \* ByteStreamDemo1

 \*/

public class ByteStreamDemo1

{

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

    {

        FileInputStream fis=new FileInputStream("abbo.txt");

        int r=0;

        //r=0

        while (   ( r=fis.read() )!=-1   )

        {      System.out.print((char)r);

        }

        fis.close();

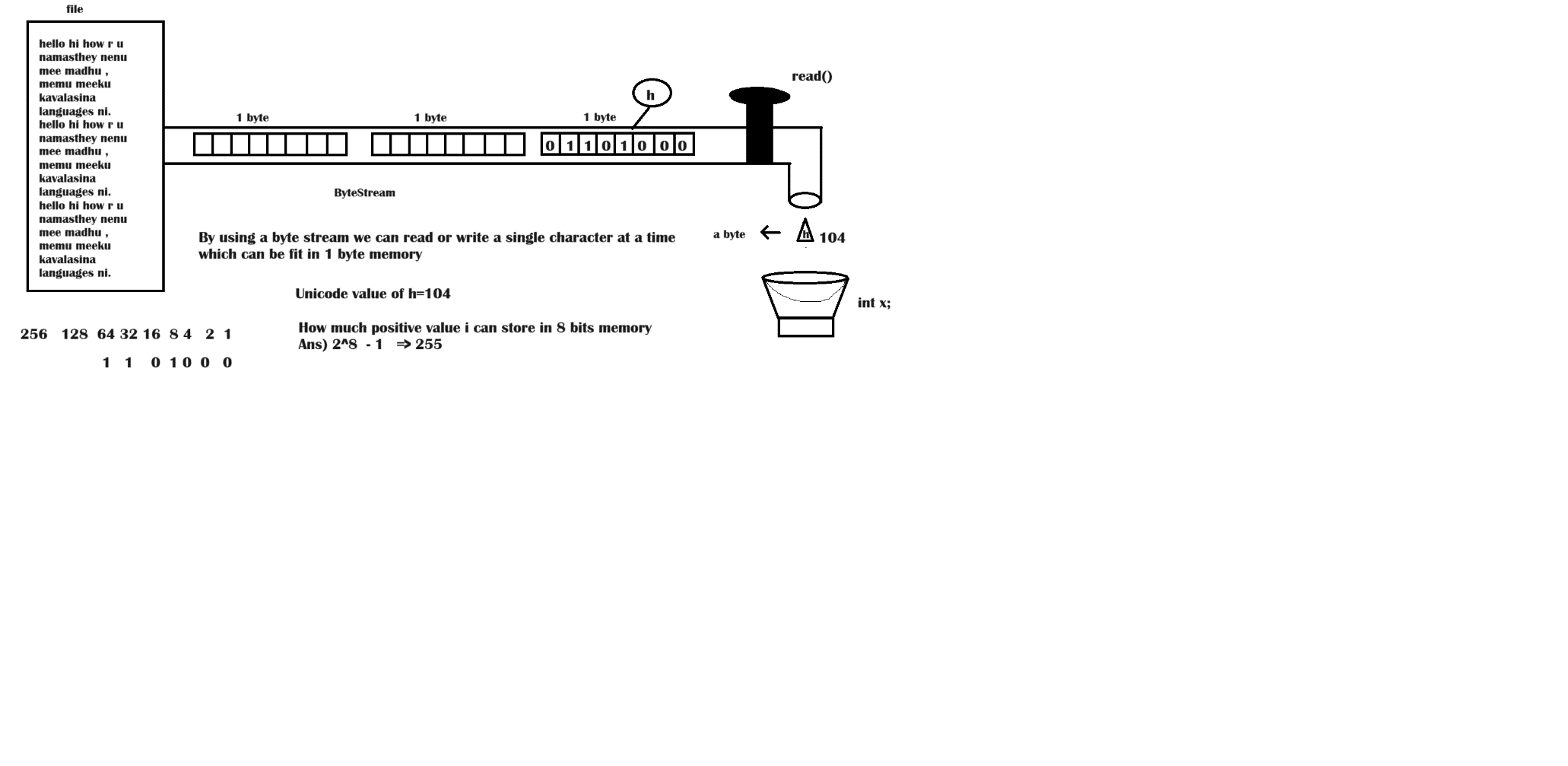
    }

}

Output:

hello hi how r u

namasthey nenu mee madhu



Same as above example by using try with resource feature

import java.io.FileInputStream;

public class ByteStreamDemo1

{

    public static void main(String[] args)

    {

        try(FileInputStream fis=new FileInputStream("abbo2.txt"))

        {

            int r=0;

            while (   ( r=fis.read() )!=-1   )

            {      System.out.print((char)r);

            }

        }catch(Exception ex)

        {   ex.printStackTrace();

        }

    }

}

Example to write the data to a file using byte stream FileOutputStream

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

public class ByStreamsDemo2

{

    public static void main(String[] args)

    {

        String fileName="perk.txt";

        String data="Mini Treats Store it in yur cabinet. raid it at your will\nEat it. Lock it. Repeat.";

        try(FileOutputStream fos=new FileOutputStream(fileName,true))

        {

            for(int i=0;i<data.length();i++)

            {//i=0,1

                fos.write(data.charAt(i));

            }

            System.out.println("Write completed into perk.txt file");

        }catch(Exception ex)

        {

            ex.printStackTrace();

        }

    }

}

**Types of Streams:**

The **java.io** package contains large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

Streams

Byte Streams

Character Streams

Input Stream

Output Stream

Reader

Writer

* **Bytestream:** classes provide support for handling I/O operations on bytes.
* **Characterstream:** classes provide support for managing I/O operations on characters.

**What is stream?**

A stream is a pipe through which we can read or write the data from one place to another place.

**Byte Stream Classes:**

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Since the streams are unidirectional, they can transmit bytes in only one direction and therefore java provides two kinds of byte stream classes:

* **InputStream** classes: all input stream classes are the child classes of **java.io.InputStream** class.
* **OutputStream** classes: all output stream classes are the child classes of **java.io.OutputStream** class.

Input Stream

String Buffer InputStream

File Input Stream

Byte Array Input Stream

**Filter Input Stream**

Sequence Input Stream

Object Input Stream

Piped Input Stream

Line Number Input Stream

Data Input Stream

Buffered Input Stream

Pushback Input Stream

Output Stream

ByteArrayOutputStream

File Output Stream

**FilterOutputStream**

ObjectOutputStream

PipedOutputStream

DataOutputStream

BufferedOutputStream

PrintStream

**Character Stream Classes:**

Character stream classes were not a part of the language when it was released in 1995. They were added later when the version 1.1 was announced, character streams can be used to read and write 16-bit Unicode characters. Like byte streams, there are two kinds of character stream classes, namely.

* **Reader stream** classes: all the reader stream classes are the child classes of the **java.io.Reader** class.
* **Writer stream** classes.: All the Writer streams classes are the child classes of the **java.io.Writer** class.

Reader

PipedReader

CharArrayReader

StringReader

BufferedReader

InputStreamReader

FilterReader

LineNumberedReader

FileReader

PushbackReader

Writer

StringWriter

CharArrayWriter

PipedWriter

OutputStreamWriter

FilterWriter

BufferedWriter

PrintWriter

FileWriter

**Character Streams**

|  |
| --- |
| These streams are used to read or write the data character(2bytes) by character(2bytes)  Character streams are of 2 types   1. ReaderStreams 2. WriterStreams |

**Note:** In Java streams work in a single direction. So, stream is called as unidirectional in Java.

Example to read the data from file character by character using a character stream

import java.io.FileReader;

public class CharacterStream1

{

    public static void main(String[] args)

    {

        try(FileReader fr=new FileReader("perk.txt"))

        {

            int r=0;

            while(   (r=fr.read())!=-1  )

            {   System.out.print((char)r);

            }

        }catch(Exception ex)

        {

            ex.printStackTrace();

        }

    }

}

Example to read and write the data by using a character stream

import java.io.FileReader;

import java.io.FileWriter;

public class CharacterStream1

{

    public static void main(String[] args)

    {

        try(FileReader fr=new FileReader("perk.txt");FileWriter fw=new FileWriter("minitreats.txt"))

        {

            int r=0;

            while(   (r=fr.read())!=-1  )

            {   //System.out.print((char)r);

                fw.write(r);

            }

        System.out.println("Read and Write completed");

        }catch(Exception ex)

        {

            ex.printStackTrace();

        }

    }

}

Why character streams?

|  |
| --- |
| The draw back of byte streams is we can read or write characters which are in a range of 0 to 255 only so we can’t read or write all the characters existed in Unicode character set.  So, if you want to read or write all the characters existed in Unicode character set, we have to use a character stream. |

System.in

|  |
| --- |
| * 1. System is a class   2. In is a static field existed in System class   3. In is a InputStream type variable which contains BufferedInputStream class object.   4. In is a byte stream and input stream which is used to read the data byte by byte from standard input device called keyboard |

System class imagination

|  |
| --- |
| public final class java.lang.System  {  static InputStream in=new java.io.BufferedInputStream();  static PrintStream out=new java.io.PrintStream();  static PrintStream err=new java.io.PrintStream();  } |

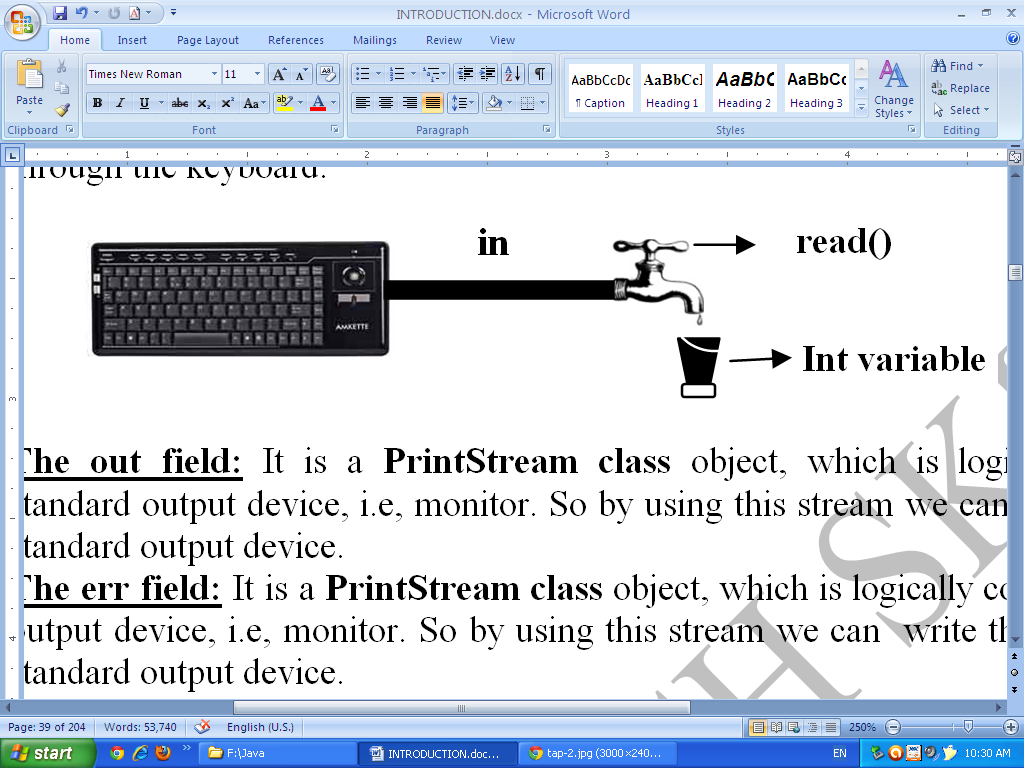
**Read the data from keyboard:**

To read the data from keyboard we need a stream which is logically connected to the standard input device (keyboard). Do we have that stream? Yes we have that stream in System class.

**Do you know about System class?**

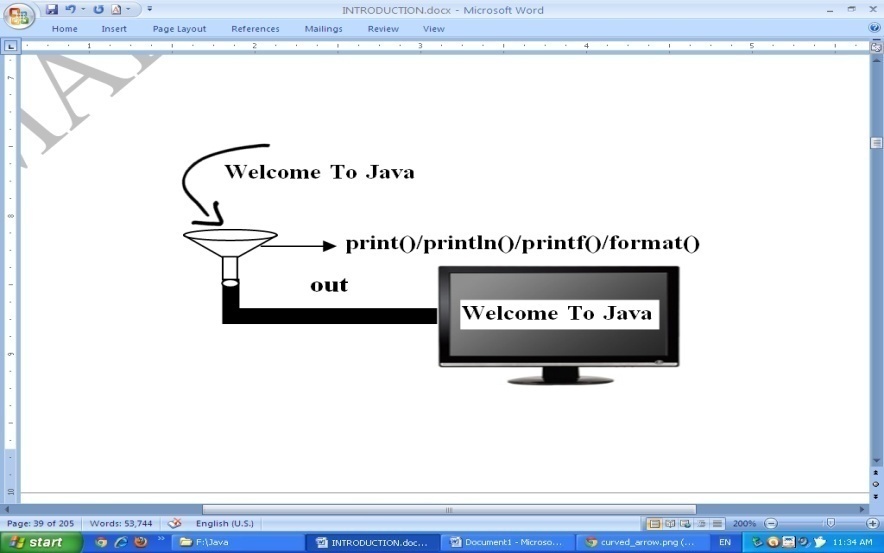
**System is a predefined** class existed in the **java.lang** package, and it has three important fields. All these three fields are declared as static, those are given below:

* **The in field:** It is an **InputStream** class object, which is logically connected to the standard input device, i.e, and keyboard. So by using this stream we can read the data given through the keyboard.

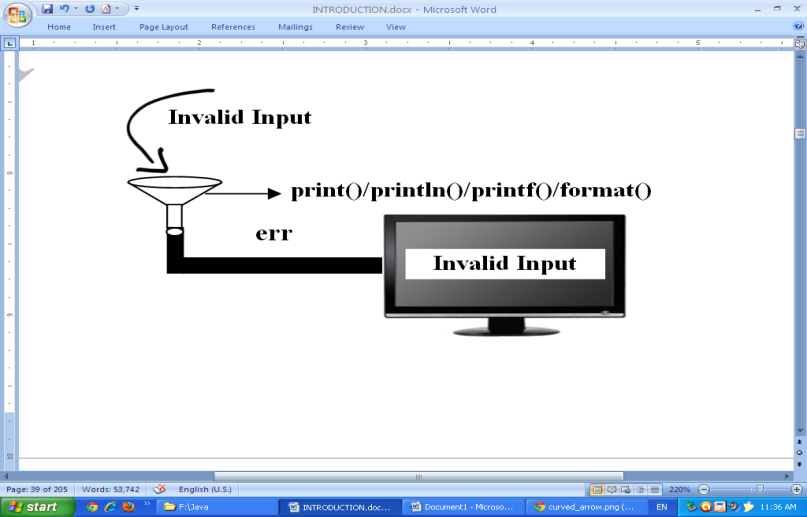


­­

* **The out field:** It is a **PrintStream class** object, which is logically connected to the standard output device, i.e, and monitor. So by using this stream we can write the output to the standard output device.



* **The err field:** It is a **PrintStream class** object, which is logically connected to the standard output device, i.e, and monitor. So by using this stream we can write the error-messages to the standard output device.



Now I will read data using the stream called **“in”** observe the below example

Example to read a single character

public class ProgramToReadChar

{

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

    {

        System.out.print("orey eeriga oka character enter cheskoraa:\t");

        int r=System.in.read();

        System.out.println(r);

    }

}

Output:

orey eeriga oka character enter cheskoraa: a

97

DataInputStream

|  |
| --- |
| * It is a filter stream and byte stream which is used to read the data from another byte stream. * It has a method called readLine() by using it we can read total line at a time. |

Example to read the total line at a time

import java.io.DataInputStream;

public class ProgramToReadChar

{

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

    {

        DataInputStream dis=new DataInputStream(System.in);

        System.out.print("Satwiko enter cheyraaa:\t");

        String line=dis.readLine();

        System.out.println("Given input:\t"+line);

    }

}

**Write a program to add two number**

Note: read the numbers using readLine() method of DataInputStream class

import java.io.DataInputStream;

public class ProgramToReadChar

{

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

    {

        DataInputStream dis=new DataInputStream(System.in);

        System.out.print("Enter int value:\t");

        //int a <= 10  <-  parseInt()  <- "10" <-  dis.readLine()  <- 10  <-keyboard

        int a=Integer.parseInt(   dis.readLine()  );  //10

        System.out.print("Enter Anoher int value:\t");

        int b=Integer.parseInt(dis.readLine());

        int c=a+b;

        System.out.printf("%d + %d = %d",a,b,c);

    }

}

Output:

Enter int value: 100

Enter Anoher int value: 20

100 + 20 = 120

Example to read integer by using readLine() method of BufferedReader class

import java.io.BufferedReader;

import java.io.DataInputStream;

import java.io.InputStreamReader;

public class ProgramToReadChar

{

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

    {

        InputStreamReader isr=new InputStreamReader(System.in);

        BufferedReader br=new BufferedReader(isr);

        System.out.print("Enter int value:\t");

        //int a <= 10  <-  parseInt()  <- "10" <-  dis.readLine()  <- 10  <-keyboard

        int a=Integer.parseInt(   br.readLine()  );  //10

        System.out.print("Enter Anoher int value:\t");

        int b=Integer.parseInt(br.readLine());

        int c=a+b;

        System.out.printf("%d + %d = %d",a,b,c);

    }

}

Output:

Enter int value: 10

Enter Anoher int value: 2

10 + 2 = 12

**Serialization & Deserialization**

**What is serialization ?**

|  |
| --- |
| * 1. Process of converting object state into binary format is called as serialization |

**What is deserialization ?**

|  |
| --- |
| * 1. Process of converting binary format into object state is called as deserialization |

**What is a persistent object?**

|  |
| --- |
| It is an object which can be stored permanently in a file or database. |

**Note**: A person who writes the class has to give the permission to the JVM to perform serialization

**How we can give permission to the JVM to perform serialization?**

|  |
| --- |
| **By implementing Serializable interface** |

**Example on serialization & Deserialization**

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

class One implements Serializable

{

    int a,b;

    One(int a,int b)

    {   this.a=a;

        this.b=b;

    }

    public int getA()

    {   return a;

    }

    public int getB()

    {   return b;

    }

    public void setA(int a)

    {   this.a=a;

    }

    public void setB(int b)

    {   this.b=b;

    }

}

public class SerDeSer

{

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

    {

        One o1=new One(100,20);

        One o2=new One(10,2);

        One o3=new One(1,2);

        FileOutputStream fos=new FileOutputStream("oyabbo.txt");

        ObjectOutputStream oos=new ObjectOutputStream(fos);

        oos.writeObject(o1);//serialization

        oos.writeObject(o2);//serialization

        oos.writeObject(o3);//serialiation

        System.out.println("i have written 3 objects to oyabbo.txt file");

        FileInputStream fis=new FileInputStream("oyabbo.txt");

        ObjectInputStream ois=new ObjectInputStream(fis);

        One obj1=(One)ois.readObject(); //de-serialization

        One obj2=(One)ois.readObject(); //de-serialization

        One obj3=(One)ois.readObject(); //de-serialization

        System.out.println("First object");

        System.out.println(obj1.getA()+","+obj1.getB());

        System.out.println("Second object");

        System.out.println(obj2.getA()+","+obj2.getB());

        System.out.println("Third object");

        System.out.println(obj3.getA()+","+obj3.getB());

        System.out.println(o1==obj1);

        System.out.println(o2==obj2);

        System.out.println(o3==obj3);

    }

}

Output:

i have written 3 objects to oyabbo.txt file

First object

100,20

Second object

10,2

Third object

1,2

false

false

false

**Can we create an object without calling constructor**

|  |
| --- |
| Yes, during deserialization object will be created without calling constructor. |

**­­­**

**What is the use of transient keyword?**

|  |
| --- |
| If u declare a field of a class as transient that will not support serialization. So If you don’t want to serialize some fields of a class declare those fields as transient. |

**2.Implement Serializable and get a chance to modify the default protocol**

In the above class we just uses the standard mechanism and demonstrates serialization using Emp and Student classes.

**Customize Java Serialization**

We know Serializable is a java marker interface. When a class implements Serializable interface it gives information to the [JVM](http://javapapers.com/core-java/differentiate-jvm-jre-jdk-jit/)that the instances of these classes can be serialized. Along with that, there is a special note to the JVM which tells that *look for following two methods in the class that implements Serializable. If found invoke them and continue with serialization process else directly follow the standard serialization protocol.*

**Those methods are given below**

* private void writeObject(ObjectOutputStream out) throws IOException;
* private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;

In the below example I have modified the Emp class by including these two methods . This is not overriding or overloading methods and this is a mechanism provided by serialization.

**Note:**

|  |
| --- |
| These two included methods are declared private but JVM can access the private methods of an object. There is no change to the class that does the serialization and de-serialization |

**Example: OISOOS2.java**

|  |
| --- |
| import java.io.\*;  import static java.lang.System.\*;  class Emp implements Serializable  { private int eno;  private String ename;  Emp(int eno,String ename)  {this.eno=eno;  this.ename=ename;  }  public void setEno(int eno)  {this.eno=eno;  }  public int getEno()  {return eno;  }  public void setEname(String ename)  {this.ename=ename;  }  public String getEname()  {return ename;  }  public String toString()  {return "Eno:=\t"+eno+"\nEname:\t"+ename;  }  private void writeObject(ObjectOutputStream out) throws IOException  { setEno(enc(this.eno));  setEname(enc(this.ename));  out.defaultWriteObject();  }  int enc(int n)  {  n=((n+n)+1)+271;  return n;  }  String enc(String str)  { String res="";  for(int i=0;i<str.length();i++)  {res=res+(char)(str.charAt(i)+(i\*10));  }  return res;  }  int dec(int n)  {  n=((n-271)-1)/2;  return n;  }  String dec(String str)  { String res="";  for(int i=0;i<str.length();i++)  {res=res+(char)(str.charAt(i)-(i\*10));  }  return res;  }  private void readObject(ObjectInputStream in)throws IOException, ClassNotFoundException  {in.defaultReadObject();  setEno(dec(this.eno));  setEname(dec(this.ename));  }  }  class OOSOIS2  {  public static void main(String args[])throws Exception  { Emp e=new Emp(101,"Madhu");  FileOutputStream fos=new FileOutputStream("Objets.data");  ObjectOutputStream oos=new ObjectOutputStream(fos);  oos.writeObject(e);  oos.close();  fos.close();  FileInputStream fis=new FileInputStream("Objets.data");  ObjectInputStream ois=new ObjectInputStream(fis);  out.println(ois.readObject());  ois.close();  fis.close();  }  } |

**Disadvantage:**

If we want to implement our own serialization mechanism by implementing the **Serializable** interface(yes it is also possible by defining **writeObject** and **readObject** methods) we don't need to override or implement any method.

The JVM calls the serialization methods from our class **using reflection**. In early JVM implementations **reflection performance was** kind of **slow,** so to overcome this problem the **Externalizable** interface was introduced.

**3.Implement Externalizable and write your own protocol to implement serailization.**

**Externalizable** is an interface that enables you to define custom rules and your own mechanism for serialization. Implementing the **Externalizable** interface means that we must override some of its methods, namely the **writeExternal** and **readExternal** methods. These methods will be called when you serialize (or deserialize) a given instance.

***The Externalizable interface is defined as:***

**public interface Externalizable extends Serializable**

**The method of Externalizable interface**

public void writeexternal (ObjectOutput out) throws IOException;

public void readexternal (ObjectInput in) throws IOException, ClassNotFoundException;

**writeExternal():**

a writeExternal() method for storing its state during serialization.

**readExternal():**

a readExternal() method for restoring its state during deserialization.

**Example: OISOOS.java**

|  |
| --- |
| import java.io.\*;  import static java.lang.System.\*;  class Emp implements Externalizable  { private int eno;  private String ename;  Emp(int eno,String ename)  {this.eno=eno;  this.ename=ename;  }  public Emp()  {out.println("Called...");  }  public void setEno(int eno)  {this.eno=eno;  }  public int getEno()  {return eno;  }  public void setEname(String ename)  {this.ename=ename;  }  public String getEname()  {return ename;  }  public String toString()  {return "Eno:=\t"+eno+"\nEname:\t"+ename;  }  @Override  public void writeExternal(ObjectOutput out) throws IOException  { out.writeInt(enc(eno));  out.writeObject(enc(ename));  }  @Override  public void readExternal(ObjectInput in) throws IOException,ClassNotFoundException  { eno = dec(in.readInt());  ename =dec((String) in.readObject());  }  int enc(int n)  {  n=((n+n)+1)+271;  return n;  }  String enc(String str)  { String res="";  for(int i=0;i<str.length();i++)  {res=res+(char)(str.charAt(i)+(i\*10));  }  return res;  }  int dec(int n)  {  n=((n-271)-1)/2;  return n;  }  String dec(String str)  { String res="";  for(int i=0;i<str.length();i++)  {res=res+(char)(str.charAt(i)-(i\*10));  }  return res;  }  }  class OOSOIS3  { public static void main(String args[])throws Exception  { Emp e=new Emp(102,"Kiran");  FileOutputStream fos=new FileOutputStream("Objets.data");  ObjectOutputStream oos=new ObjectOutputStream(fos);  oos.writeObject(e);  oos.close();  fos.close();  FileInputStream fis=new FileInputStream("Objets.data");  ObjectInputStream ois=new ObjectInputStream(fis);  out.println(ois.readObject());  ois.close();  fis.close();  }  } |

**Concepts we have covered till now**

* + 1. **Introduction**
    2. **Features**
    3. **Jdk installations**
    4. **VSCode installation**
    5. **JCL**
    6. **Java API**
    7. **Escape sequences**
    8. **Naming conventions**
    9. **Compilation**
    10. **Execution**
    11. **Data types**
    12. **If**
    13. **If..else**
    14. **If..else..if**
    15. **Nested if**
    16. **Switch**
    17. **Array**
    18. **1D arrays**
    19. **2d arrays**
    20. **3d arrays**
    21. **Jagged arrays**
    22. **For loop**
    23. **For each**
    24. **While**
    25. **Do..while..**
    26. **Functions(methods)**
    27. **Class**
    28. **Object**
    29. **Instance variables**
    30. **Static variables**
    31. **Instance methods**
    32. **Static methods**
    33. **Instance block**
    34. **Static block**
    35. **Constructors**
    36. **This keyword**
    37. **String constant pool**
    38. **Accessor and mutator methods**
    39. **Inheritance**
    40. **Super()**
    41. **Method overloading**
    42. **Method overriding**
    43. **Upcasting & Down casting**
    44. **Static and dynamic binding**
    45. **Compile time and runtime Polymorphism**
    46. **Abstract classes**
    47. **Interfaces**
    48. **Inner classes**
    49. **Anonymous inner classes**
    50. **Functional interfaces**
    51. **Lambda expressions**
    52. **Singleton**
    53. **Immutable object**
    54. **Creating record using contextual keyword called ‘record’**
    55. **Exception Handling**
    56. **Try**
    57. **Catch**
    58. **Finally**
    59. **Throw**
    60. **Throws**
    61. **Nested try**
    62. **Types of exceptions**
    63. **User defined exceptions**
    64. **Mult catch exceptions**