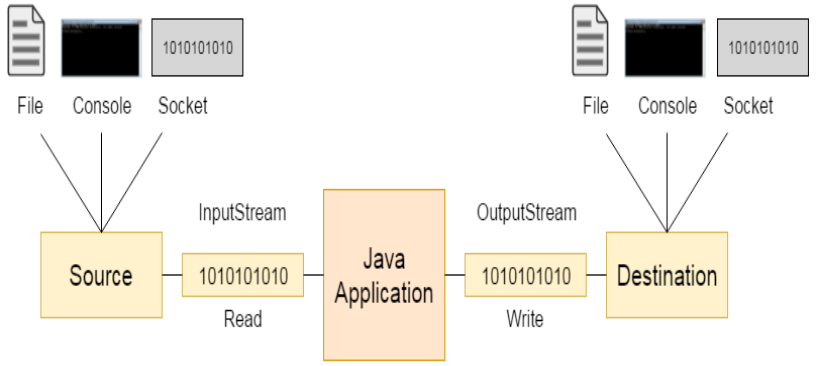
Input/Output Fundamentals

I/O Streams

- Java performs I/O through **Streams**.
- *Stream* literally means continuous flow, and I/O stream in Java refers to the flow of bytes between an input source and output destination.
- The type of sources or destination can be anything that contains, generates, or consumes data.
- Alot goes on behind the scenes, even if it is seemingly a simple I/O flow from one end to another.
- Implementing them from scratch is by no means simple and needs to go through the rigor of extensive coding.
- Java Stream APIs handle these complexities, giving developers an open space to concentrate on their productive ends rather than brainstorm on the intricacies of I/O processing.
- One just needs to understand the right use of the API interfaces, objects, and methods and let it handle the intricacies on their behalf.



Stanard Streams

- All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen.
- Java provides the following three standard streams –
  - **Standard Input**
    - A keyboard is used as standard input stream and represented as **System.in**.
  - **Standard Output**
    - A computer screen is used for standard output stream and represented as **System.out**.
  - **Standard Error**
    - A computer screen is used for standard error stream and represented as **System.err**.

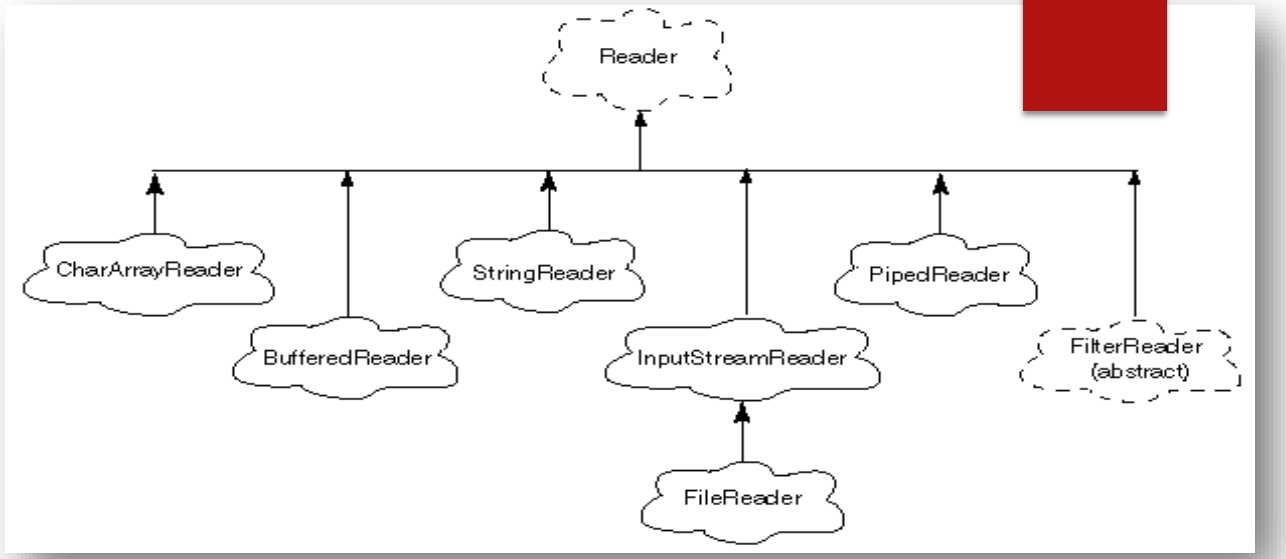
Stream Types

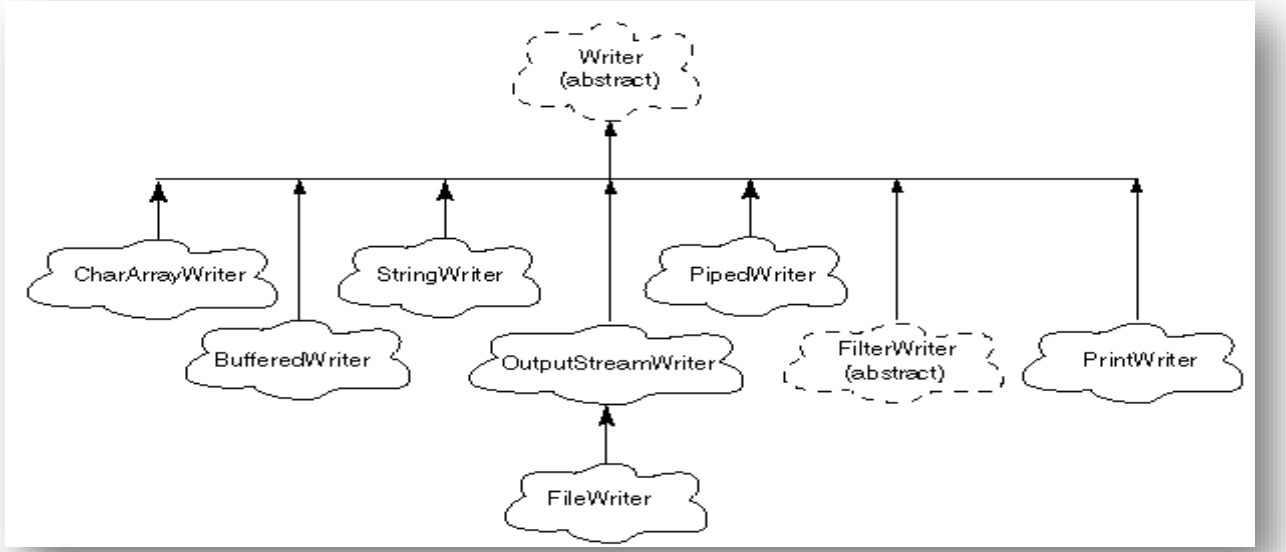
- Byte-oriented streams.
  - Handle data in the form of bits and bytes
  - Byte streams are used to handle anycharacters (text), images, audio and video files For example, to store an image file (gif orjpg), we should go for a byte stream
- Character-oriented streams
  - Handle data in the form of characters
  - Character or textstreams can always store and retrieve data in the form of characters (or text) only
  - It means text streams are more suitable for handling text files like the ones we create in Notepad
  - They are not suitable to handle the images, audio or video files To handle data in the form of 'text'
  - Transforms data from/to 16 bit Java char used inside programs to UTF format used externally

Reader and Writer Classes

- **Reader** is an abstract class from which all character-oriented input streams are derived
- All these streams deliver 16-bit **char** data to a program

- **Writer** is an abstract class from which all character-oriented output streams are derived
- All these streams receive 16-bit char data from a program, and send it to another destination, which may use a different format (such as UTF format on a disk file)

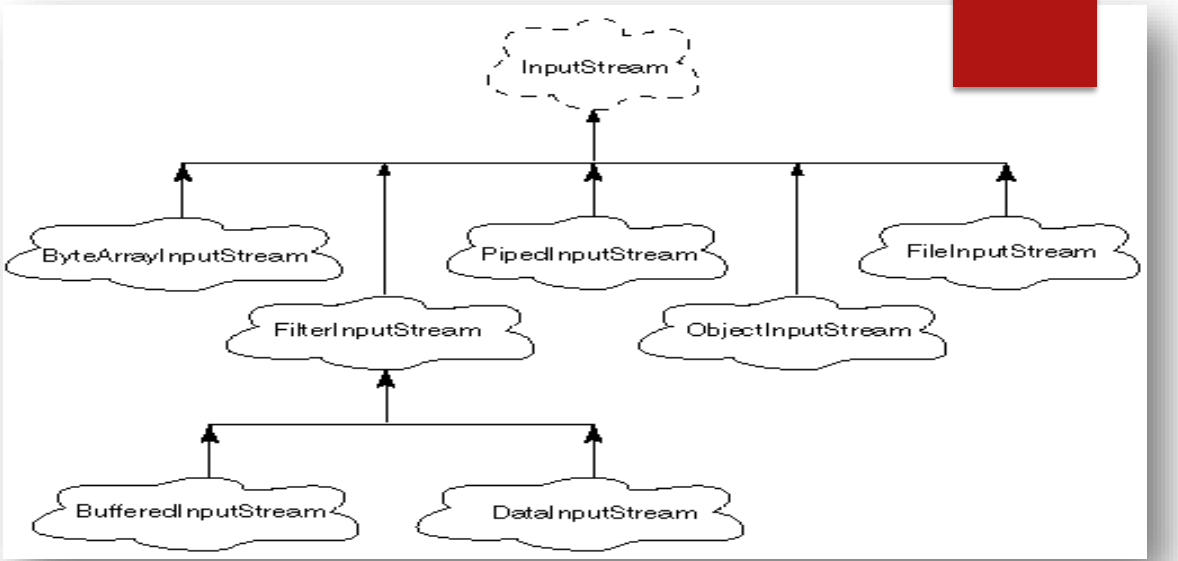


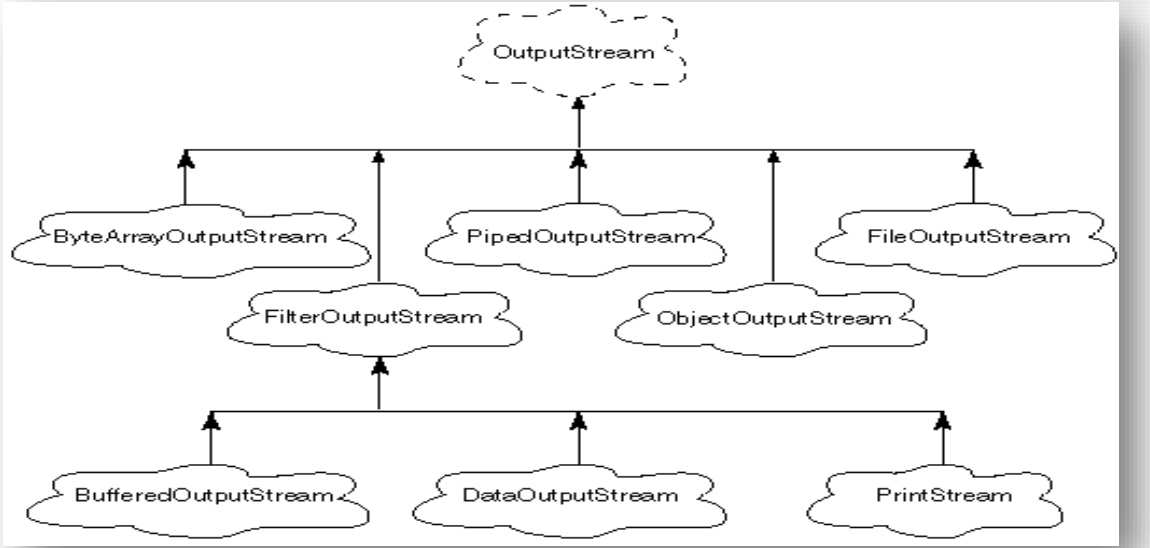


InputStream and OutputStream

- **InputStream** is an abstract class from which all ***byte***-oriented input streams are derived
- These streams are aimed at delivering data to a program in groups of 8-bit bytes.
- For example, if a disk file contains 32-bit **int** data, data can be delivered to the program in 4-byte groups in the same format as Java primitive type **int**

- **OutputStream** is an abstract class from which all byte-oriented output streams are derived
- These streams are aimed at writing groups of 8-bit bytes to output destinations





Classes we are going to dicuss

- Input and OutputStream derived Classes
  - FileInputStream and FileOutputStream
  - BufferedInputStream and BufferedOutputStream
  - DataInputStream and DataOutputStream
  - ObjectInputStream and ObjectOutputStram
- Reader and Writer derived Classes
  - FileReader and FileWriter
  - BufferedReader and BufferedWriter

FileInputStream and FileOutputStream

- Streams for reading and writing data to and from files
- Common Methods used:
  - read() - to read bytes from a file
  - write(byte[] b) - to write bytes to a file
- First Program creates file named ***mydata.txt*** and second reads the data from that file

```java
import java.io.*;
class FileOutputStreamExample{
   public static void main(String args[])
   {
      FileInputStream is = null;
      try {
         is = new FileInputStream("mydata.txt");
         int next;
         while((next=is.read())!=-1) {
            System.out.println("next = " + (char)next);
         }
         is.close();
      } catch(IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

mydata.txt

```
ABCDEFGHIJ
```

```java
import java.io.*;
class FileStreamExample{
   public static void main(String args[])
   {
      FileOutputStream os = null;
      try {
         os = new FileOutputStream("mydata.txt");
         for(int i = 65; i < 75; i++) {
            os.write(i);
         }
         os.close();
      }
      catch(IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

```
next = A
next = B
next = C
next = D
next = E
next = F
next = G
next = H
next = I
next = J
```

BufferedInputStream and BufferedOutputStream

- Buffered input streams read more data than they initially need into a buffer (an internal array of bytes).
- When the stream's read() methods are invoked, the data is removed from the buffer rather than the underlying stream.
- When the buffer runs out of data, the buffered stream refills its buffer from the underlying stream.
- Likewise, buffered output streams store data in an internal byte array until the buffer is full or the stream is flushed; then the data is written out to the underlying output stream in one swoop.
- Constructors

BufferedInputStream(InputStream in)

BufferedInputStream(InputStream in, int size)

BufferedOutputStream(OutputStream out)

BufferedOutputStream(OutputStream out, int size)

```java
import java.io.*;
class BufferedFileStreamExample{
   public static void main(String args[])
   {
      FileOutputStream os = null;
      try {
         os = new FileOutputStream("mydata.txt");
         BufferedOutputStream bs=new BufferedOutputStream(os);
         for(int i = 65; i < 75; i++) {
            bs.write(i);
         }
         bs.close();
         os.close();
      }
      catch(IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

mydata.txt

```
ABCDEFGHIJ
```

```java
import java.io.*;
class BufferedFileInputStreamExample{
   public static void main(String args[])
   {
      FileInputStream is = null;
      try {
         is = new FileInputStream("mydata.txt");
         BufferedInputStream bs=new BufferedInputStream(is);
         int next;
         while((next=bs.read())!=-1) {
            System.out.println("next = " + (char)next);
         }
         bs.close();
         is.close();
      } catch(IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

```
next = A
next = B
next = C
next = D
next = E
next = F
next = G
next = H
next = I
next = J
```

FileReader and FileWriter

- Streams for reading and writing data to and from files
- Common Methods used:
  - read() - to read characters from a file
  - write(char c)
  - write(String s)
- First Program creates file named ***myfile.txt*** and second reads the data from that file

```java
import java.io.*;
class FileWriterEx{
   public static void main(String args[])
   {
      FileWriter fw = null;
      try {
         fw = new FileWriter("myfile.txt");
         fw.write("creating a new file \nwriting next line \n");
         fw.close();
      }
      catch(IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

```java
import java.io.*;
class FileReaderEx{
   public static void main(String args[])
   {
      FileReader fr = null;
      try {
         fr = new FileReader("myfile.txt");
         int next;
         while((next=fr.read())!=-1) {
            System.out.print((char)next);
         }
         fr.close();
      } catch(IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Output:

```
creating a new file
writing next line
```

BufferedReader and BufferedWriter

- Common Methods used:
  - read() - to read characters from a file
  - read(char[] cbuf, int off, int len) - read portion of an array
  - readLine() - reads string
  - write(char c)
  - write(String s)
  - write(char[] cbuf, int off, int len)
  - write(String s, int off, int len)

```java
import java.io.*;
class BufferedFileWriterEx{
    public static void main(String args[])
    {
        FileWriter fw = null;
        try {
            fw = new FileWriter("myfile.txt");
            BufferedWriter bw=new BufferedWriter(fw);
            bw.write("creating a new file \nwriting next line");
            bw.newLine();
            bw.close();
            fw.close();
        }
        catch(IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

```java
import java.io.*;
class BufferedFileReaderEx{
    public static void main(String args[])
    {
        FileReader fr = null;
        try {
            fr = new FileReader("myfile.txt");
            BufferedReader br=new BufferedReader(fr);
            String nextLine;
            while((nextLine=br.readLine())!=null) {
                System.out.println(nextLine);
            }
            br.close();
            fr.close();
        } catch(IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Output:

```
creating a new file
writing next line
```

DataInputStream and DataOutputStream

- These classes provide methods to read and write primitive data in a machine-independent way.
- Some Common Methods:
  - read()
  - readInt()
  - readDouble()
  - readBoolean()
  - readChar()
  - readUTF()
  - write(int i)
  - writeInt(int i)
  - writeChars(String s)
  - writeUTF(String s)
  - writeBoolean(boolean b)

```java
import java.io.*;
class DataOutputStreamEx{
   public static void main(String args[])
   {
      DataOutputStream os = null;
      try {
         os = new DataOutputStream(new FileOutputStream("bins.txt"));
         os.writeDouble(3.14);
         os.writeBoolean(true);
         os.writeInt(42);
         os.writeChar('q');
         os.close();
      } catch(IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

```java
import java.io.*;
class DataInputStreamEx{
   public static void main(String args[])
   {
      DataInputStream is = null;
      try {
         is = new DataInputStream(new FileInputStream("bins.txt"));
         System.out.println("next = " + is.readDouble());
         System.out.println("next = " + is.readBoolean());
         System.out.println("next = " + is.readInt());
         System.out.println("next = " + is.readChar());
         is.close();
      } catch(IOException e) {
         System.err.println(e.getMessage());
      }
   }
}
```

Output:

```
next = 3.14
next = true
next = 42
next = q
```

ObjectInputStream and ObjectOutputStream

- These classes provide methods to read and write objects from and to streams.
- In order to read and write objects, the class defining those objects must implement the ***Serializable*** Interface
- Some Common Methods:
  - readObject()
  - writeObject(Object o)

Output:

```
Ali
40
Hamza
30
End of File!
```

```java
public class ObjectStreamExample {

    public static class Person implements Serializable {
        public String name = null;
        public int    age  =   0;
    }


    public static void main(String[] args) throws IOException, ClassNotFoundException {

        ObjectOutputStream objectOutputStream =
            new ObjectOutputStream(new FileOutputStream("person.bin"));

        Person person = new Person();
        person.name = "Ali";
        person.age  = 40;

        Person person1 = new Person();
        person1.name = "Hamza";
        person1.age  = 30;

        objectOutputStream.writeObject(person);
        objectOutputStream.writeObject(person1);

        objectOutputStream.close();


        ObjectInputStream objectInputStream =
            new ObjectInputStream(new FileInputStream("person.bin"));
        try{
            while(true){
                Person personRead = (Person) objectInputStream.readObject();
                System.out.println(personRead.name);
                System.out.println(personRead.age);
            }
        }
        catch(EOFException e){
            System.err.println("End of File!");
        }
        objectInputStream.close();
    }
}
```

Serialization and Deserialization

- **Serialization** is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams.
- The reverse process of creating object from sequence of bytes is called **deserialization**.
- The Java Serializable interface (java.io.Serializable is a marker interface your classes must implement if they are to be serialized and deserialized.
- That Serializable is a marker interface means that it contains no methods.
- Therefore, a class implementing Serializable does not have to implement any specific methods.
- Implementing Serializable thus just tells the Java serialization classes that this class is intended for object serialization.

Thank You!