I/O

File Handling

- **Java I/O** (Input and Output) is used *to process the input* and *produce the output*.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform **file handling in Java** by Java I/O API.
- The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

- A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
- In Java, 3 streams are created for us automatically. All these streams are attached with the console.
- **1) System.out:** standard output stream
- **2) System.in:** standard input stream
- **3) System.err:** standard error stream

Standard 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.
- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**

I/O Streams

- Byte Streams handle I/O of raw binary data.
- Character Streams handle I/O of character data, automatically handling translation to and from the local character set.
- Buffered Streams optimize input and output by reducing the number of calls to the native API.
- I/O from the Command Line describes the Standard Streams and the Console object.
- Data Streams handle binary I/O of primitive data type and String values.
- Object Streams handle binary I/O of objects.

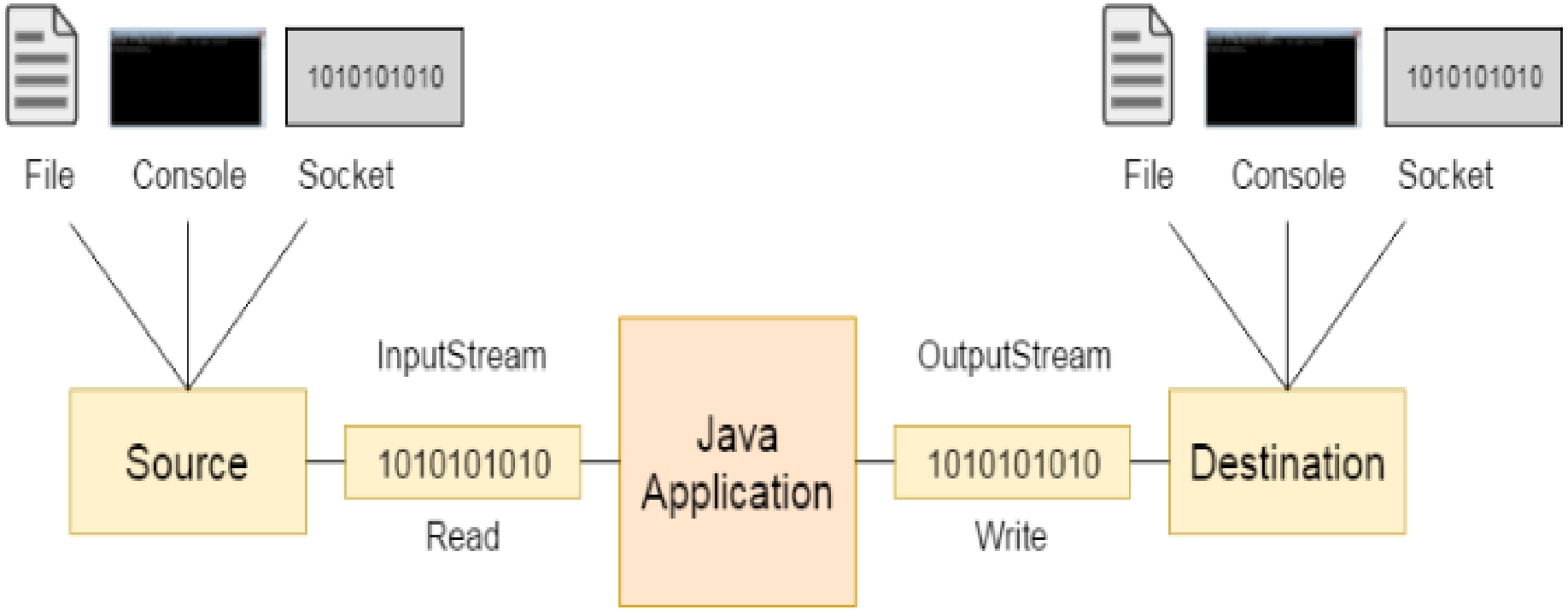
OutputStream vs InputStream

InputStream

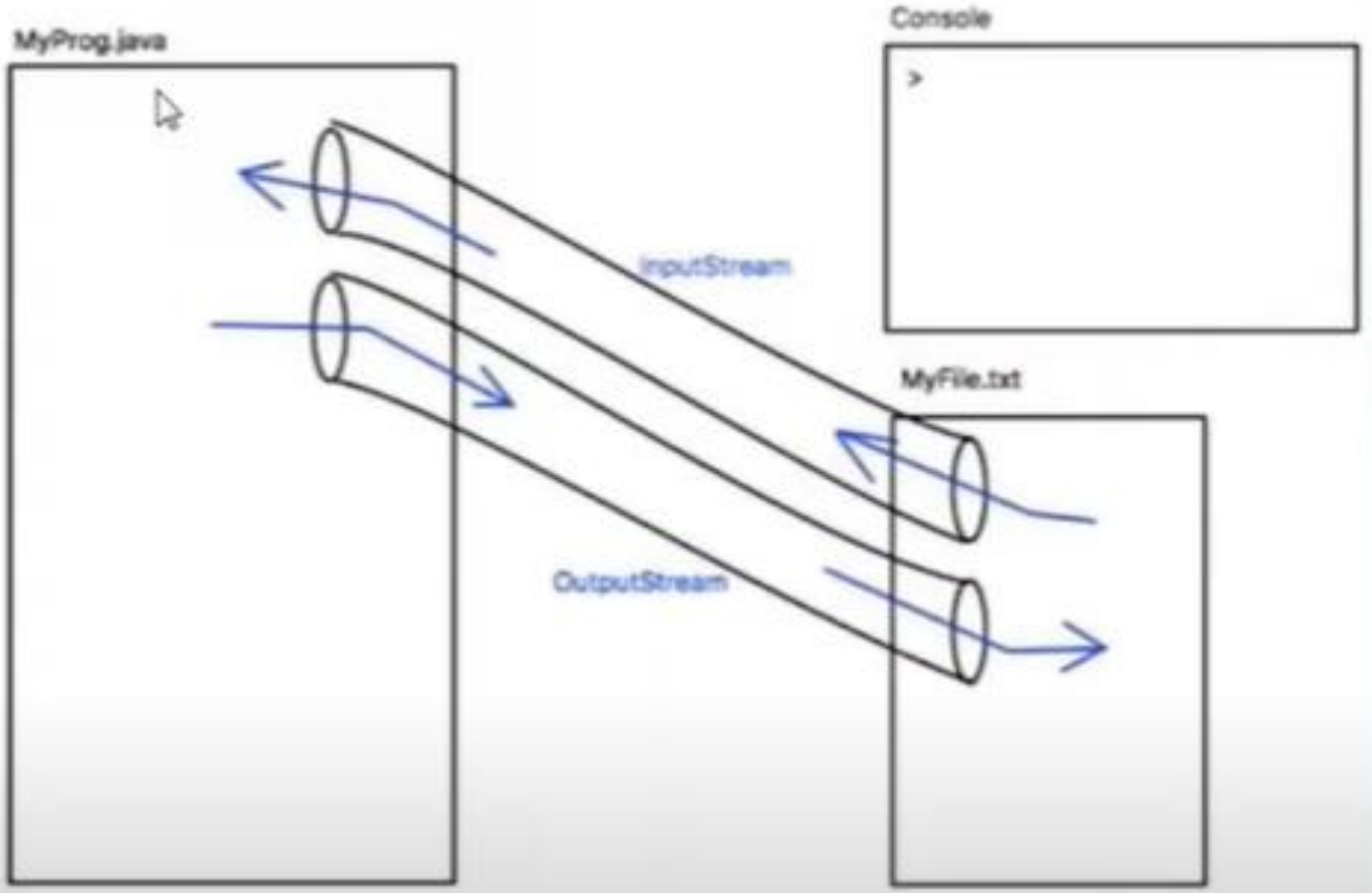
- Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

OutputStream

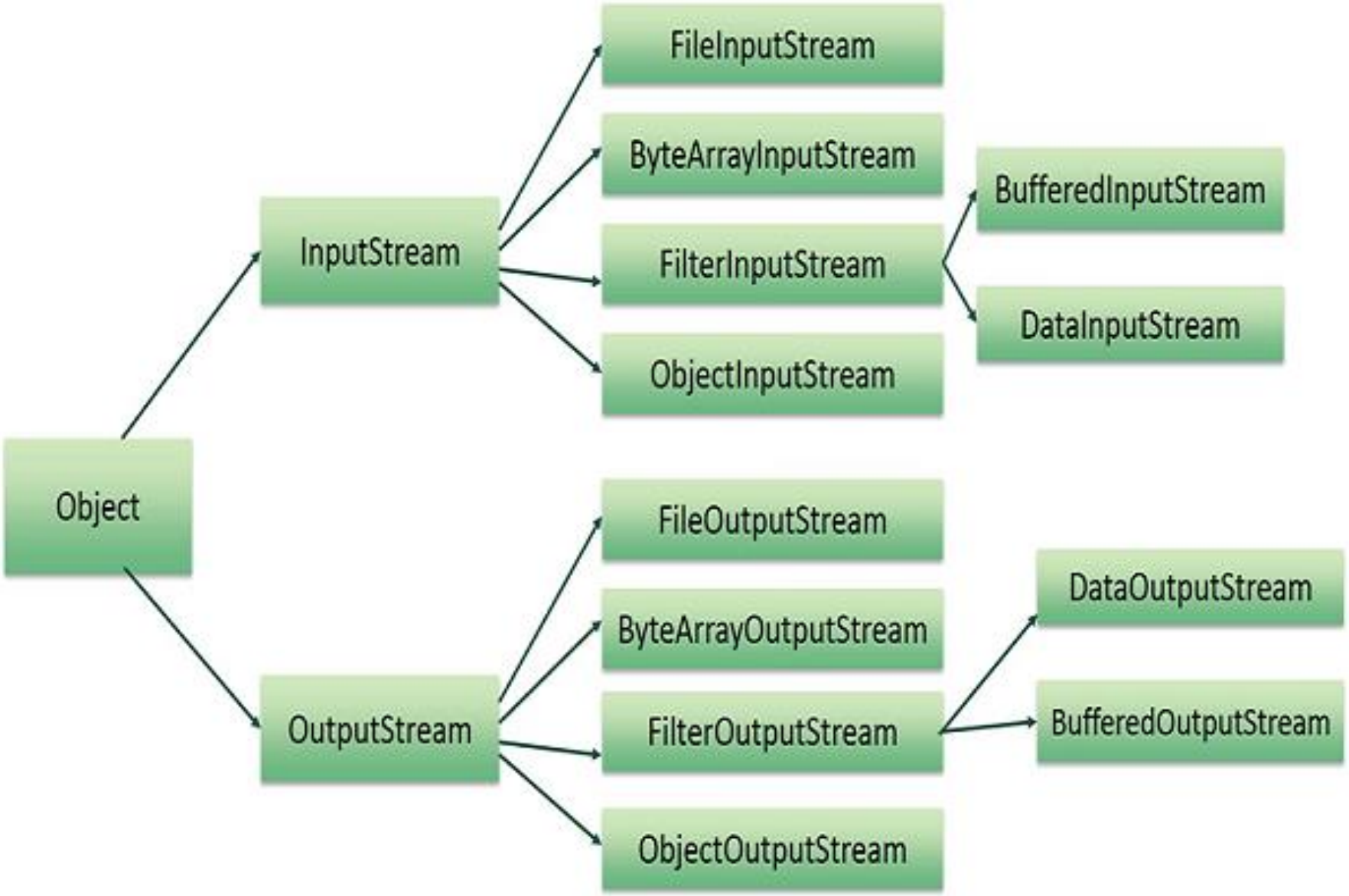
- Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.



- Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.
- **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**.


MyProg.java
Console
>
InputStream
MyFile.txt
OutputStream

Reading and Writing Files



OutputStream class

- It is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

InputStream class

- InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

FileOutputStream Class

- It is an output stream used for writing data to a file.
- To write primitive values into a file, we use FileOutputStream class (byte-oriented as well as character-oriented data) through FileOutputStream class.
- But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

```
public class FileOutputStream extends OutputStream
```

FileOutputStream class methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write ary.length bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write len bytes from the byte array starting at offset off to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the stream.
void close()	It is used to closes the file output stream.

Example 1: write byte

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
    public static void main(String args[]){
          try{
           FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
           fout.write(65);
           fout.close();
           System.out.println("success...");
          }catch(Exception e){System.out.println(e);}
    }
}
```

example 2: write string

```
import java.io.FileOutputStream;
public class FileOutputStreamExample {
    public static void main(String args[]){
          try{
            FileOutputStream fout=new FileOutputStream("C:\\sakku");
            String s="Welcome to Java Advance Technology.";
            byte b[]=s.getBytes();//converting string into byte array
            fout.write(b);
            fout.close();
            System.out.println("success...");
           }catch(Exception e){System.out.println(e);}
      }
}
```

FileInputStream Class

- Java FileInputStream class obtains input bytes from a file.
- It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. (also read character-stream data.)
- But, for reading streams of characters, it is recommended to use FileReader class.

```
public class FileInputStream extends InputStream
```

FileInputStream class methods

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to **len** bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream.

example 1: read single character

```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("D:\\testout.txt");
          int i=fin.read();
          System.out.print((char)i);

          fin.close();
        }catch(Exception e){System.out.println(e);}
       }
      }
```

example 2: read all characters

```
import java.io.FileInputStream;
public class DataStreamExample {
    public static void main(String args[]){
        try{
          FileInputStream fin=new FileInputStream("D:\\testout.txt");
          int i=0;
          while((i=fin.read())!=-1){
           System.out.print((char)i);
          }
          fin.close();
         }catch(Exception e){System.out.println(e);}
        }
       }
```

copy an input file into an output file

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

   public static void main(String args[]) throws IOException {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

BufferedOutputStream Class

- Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

```java
import java.io.*;
public class BufferedOutputStreamExample{
public static void main(String args[])throws Exception{
     FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
     BufferedOutputStream bout=new BufferedOutputStream(fout);
     String s="Welcome to VANIER.";
     byte b[]=s.getBytes();
     bout.write(b);
     bout.flush();
     bout.close();
     fout.close();
     System.out.println("success");
}
}
```

BufferedInputStream Class

```
import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("D:\\testout.txt");
    BufferedInputStream bin=new BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){
     System.out.print((char)i);
    }
    bin.close();
    fin.close();
  }catch(Exception e){System.out.println(e);}
 }
}
```

SequenceInputStream Class

- Java SequenceInputStream class is used to read data from multiple streams. It reads data sequentially (one by one).

```
import java.io.*;
class InputStreamExample {
  public static void main(String args[])throws Exception{
   FileInputStream input1=new FileInputStream("D:\\testin.txt");
   FileInputStream input2=new FileInputStream("D:\\testout.txt");
   SequenceInputStream inst=new SequenceInputStream(input1, input2);
   int j;
   while((j=inst.read())!=-1){
    System.out.print((char)j);
   }
   inst.close();
   input1.close();
   input2.close();
  }
}
```

Reads data from2 files and writes into another

```
import java.io.*;
class Input1{
  public static void main(String args[])throws Exception{
   FileInputStream fin1=new FileInputStream("D:\\testin1.txt");
   FileInputStream fin2=new FileInputStream("D:\\testin2.txt");
   FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
   SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
   int i;
   while((i=sis.read())!=-1)
   {
    fout.write(i);
   }
   sis.close();
   fout.close();
   fin1.close();
   fin2.close();
   System.out.println("Success..");
  }
}
```

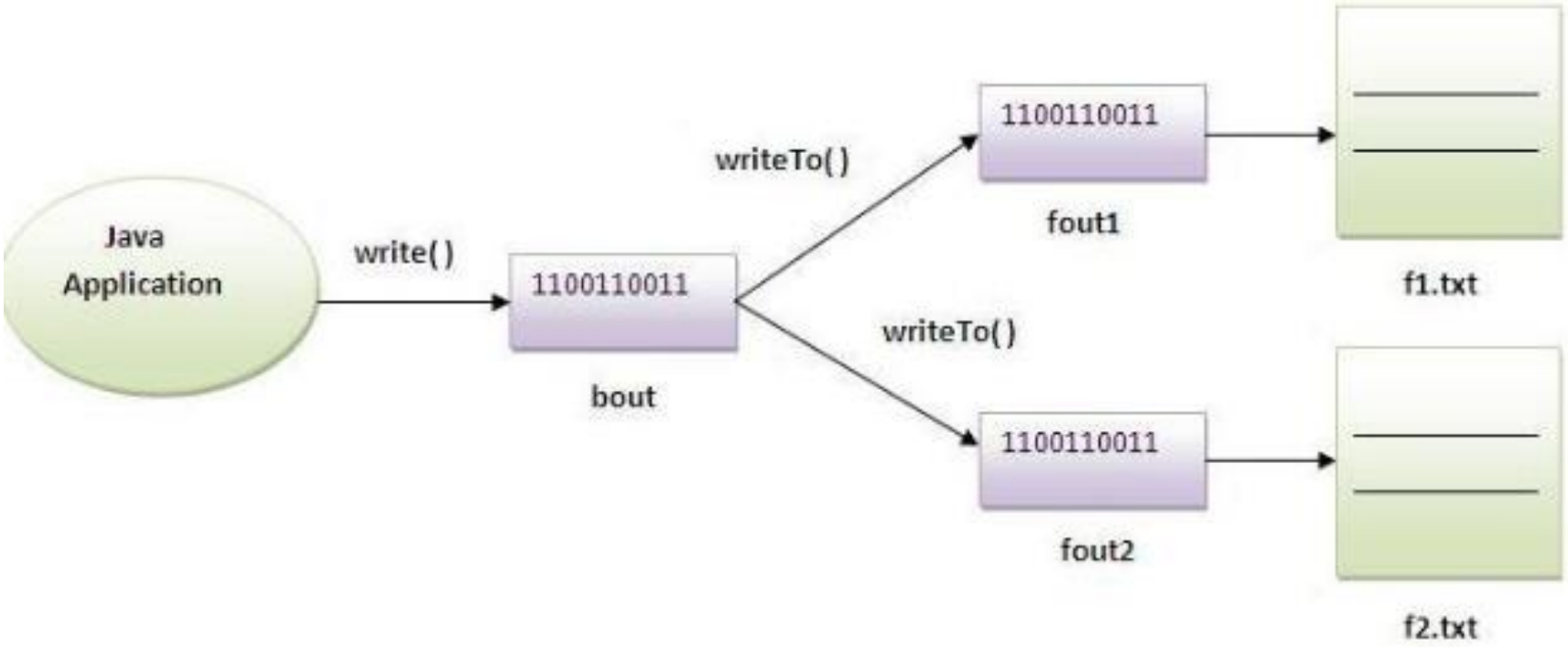
ByteArrayOutputStream Class

- ByteArrayOutputStream class is used to **write common data** into multiple files. In this stream, the data is written into a byte array which can be written to multiple streams later.
- The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.
- The buffer of ByteArrayOutputStream automatically grows according to data.

```
import java.io.*;
public class DataStreamExample {
public static void main(String args[])throws Exception{
    FileOutputStream fout1=new FileOutputStream("D:\\f1.txt");
    FileOutputStream fout2=new FileOutputStream("D:\\f2.txt");

    ByteArrayOutputStream bout=new ByteArrayOutputStream();
    bout.write(65);
    bout.writeTo(fout1);
    bout.writeTo(fout2);

    bout.flush();
    bout.close();//has no effect
    System.out.println("Success...");
 }
}
```


Java
Application
write( )
1100110011
bout
writeTo( )
1100110011
fout1
writeTo( )
1100110011
fout2
f1.txt
f2.txt

ByteArrayInputStream Class

- The ByteArrayInputStream is composed of two words: ByteArray and InputStream. As the name suggests, it can be used to read byte array as input stream.
- Java ByteArrayInputStream class contains an internal buffer which is used to **read byte array** as stream. In this stream, the data is read from a byte array.

```java
import java.io.*;
public class ReadExample {
  public static void main(String[] args) throws IOException {
    byte[] buf = { 35, 36, 37, 38 };
    // Create the new byte array input stream
    ByteArrayInputStream byt = new ByteArrayInputStream(buf);
    int k = 0;
    while ((k = byt.read()) != -1) {
      //Conversion of a byte into character
      char ch = (char) k;
      System.out.println("ASCII value of Character is:" + k + "; Special character is: " + ch);
    }
  }
}
```

DataOutputStream Class

- Java DataOutputStream class allows an application to write primitive Java data types to the output stream in a machine-independent way.
- Java application generally uses the data output stream to write data that can later be read by a data input stream.

```java
import java.io.*;
public class OutputExample {
    public static void main(String[] args) throws IOException {
        FileOutputStream file = new FileOutputStream(D:\\testout.txt);
        DataOutputStream data = new DataOutputStream(file);
        data.writeInt(65);
        data.flush();
        data.close();
        System.out.println("Success...");
    }
}
```

```
import java.io.*;
public class DataStreamExample {
  public static void main(String[] args) throws IOException {
    InputStream input = new FileInputStream("D:\\testout.txt");
    DataInputStream inst = new DataInputStream(input);
    int count = input.available();
    byte[] ary = new byte[count];
    inst.read(ary);
    for (byte bt : ary) {
      char k = (char) bt;
      System.out.print(k+"-");
    }
  }
}
```

File Reader

- Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

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

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

More Examples

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

   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

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

   public static void main(String args[]) {

      try {
         byte bWrite [] = {11,21,3,40,5};
         OutputStream os = new FileOutputStream("test.txt");
         for(int x = 0; x < bWrite.length ; x++) {
            os.write( bWrite[x] );   // writes the bytes
         }
         os.close();

         InputStream is = new FileInputStream("test.txt");
         int size = is.available();

         for(int i = 0; i < size; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      } catch (IOException e) {
         System.out.print("Exception");
      }
   }
}
```

Create directory

```
import java.io.File;
public class CreateDir {

   public static void main(String args[]) {
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);

      // Create directory now.
      d.mkdirs();
   }
}
```

Listing Directories

```
import java.io.File;
public class ReadDir {

   public static void main(String[] args) {
      File file = null;
      String[] paths;

      try {
         // create new file object
         file = new File("/tmp");

         // array of files and directory
         paths = file.list();

         // for each name in the path array
         for(String path:paths) {
            // prints filename and directory name
            System.out.println(path);
         }
      } catch (Exception e) {
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

```
import java.io.*;

public class Test {
    public static void main(String [] args) {

        // The name of the file to open.
        String fileName = "temp.txt";

        // This will reference one line at a time
        String line = null;

        try {
            // FileReader reads text files in the default encoding.
            FileReader fileReader =
                new FileReader(fileName);

            // Always wrap FileReader in BufferedReader.
            BufferedReader bufferedReader =
                new BufferedReader(fileReader);

            while((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }
```

```
            // Always close files.
            bufferedReader.close();
        }
        catch(FileNotFoundException ex) {
            System.out.println(
                "Unable to open file '" +
                fileName + "'");
        }
        catch(IOException ex) {
            System.out.println(
                "Error reading file '"
                + fileName + "'");
            // Or we could just do this:
            // ex.printStackTrace();
        }
    }
}
```

Writing Text Files in Java

```
import java.io.*;

public class Test {
    public static void main(String [] args) {

        // The name of the file to open.
        String fileName = "temp.txt";

        try {
            // Assume default encoding.
            FileWriter fileWriter =
                new FileWriter(fileName);

            // Always wrap FileWriter in BufferedWriter.
            BufferedWriter bufferedWriter =
                new BufferedWriter(fileWriter);
```

```
            // Note that write() does not automatically
            // append a newline character.
            bufferedWriter.write("Hello there,");
            bufferedWriter.write(" here is some text.");
            bufferedWriter.newLine();
            bufferedWriter.write("We are writing");
            bufferedWriter.write(" the text to the file.");

            // Always close files.
            bufferedWriter.close();
        }
        catch(IOException ex) {
            System.out.println(
                "Error writing to file '"
                + fileName + "'");
            // Or we could just do this:
            // ex.printStackTrace();
        }
    }
}
```