GUJARAT TECHNOLOGICAL UNIVERSITY

MASTER OF COMPUTER APPLICATIONS

SEMESTER: III

Subject : **4639302**

Programming in JAVA

UNIT – 5 Input and Output

Input/Output Streams: reading writing bytes, combining IO steam filers,

Text Input and Output: write text output, read text output, saving object in text format, character encoding, Reading and Writing,

Working with Files: paths, reading and writing files, creating files and directories, copying, moving and deleting files and getting file info.

Input/Output Streams:

- reading writing bytes,
- combining IO steam filers,

What is 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.

  1) System.out: standard output stream

  2) System.in: standard input stream

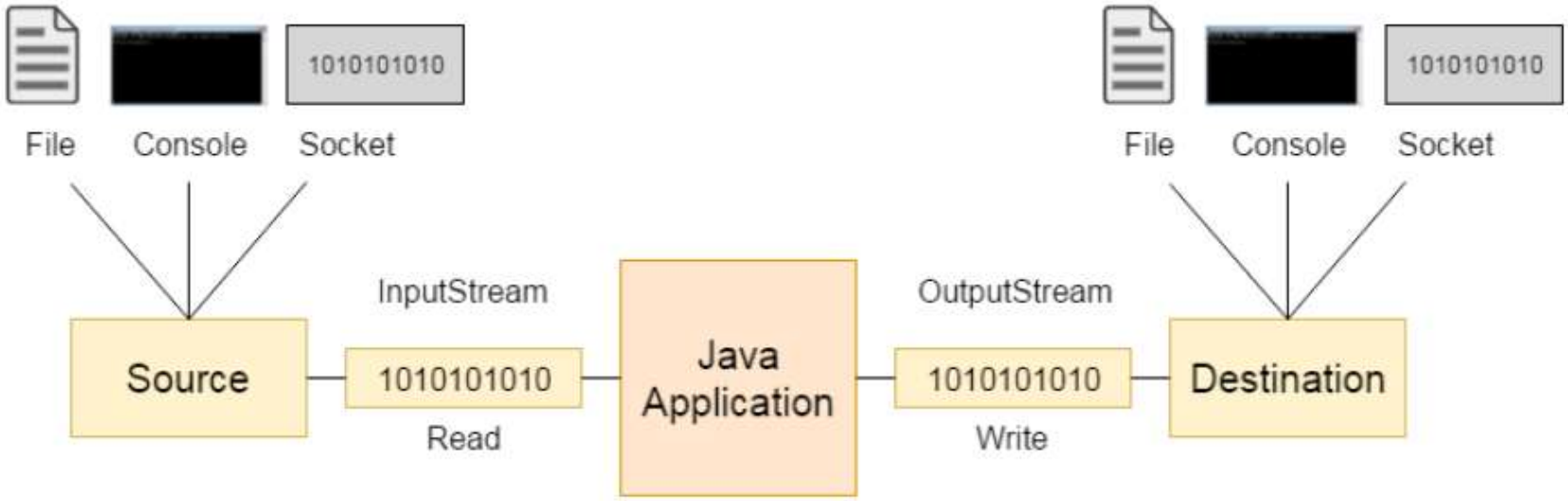
  3) System.err: standard error stream

What is I/O Stream?

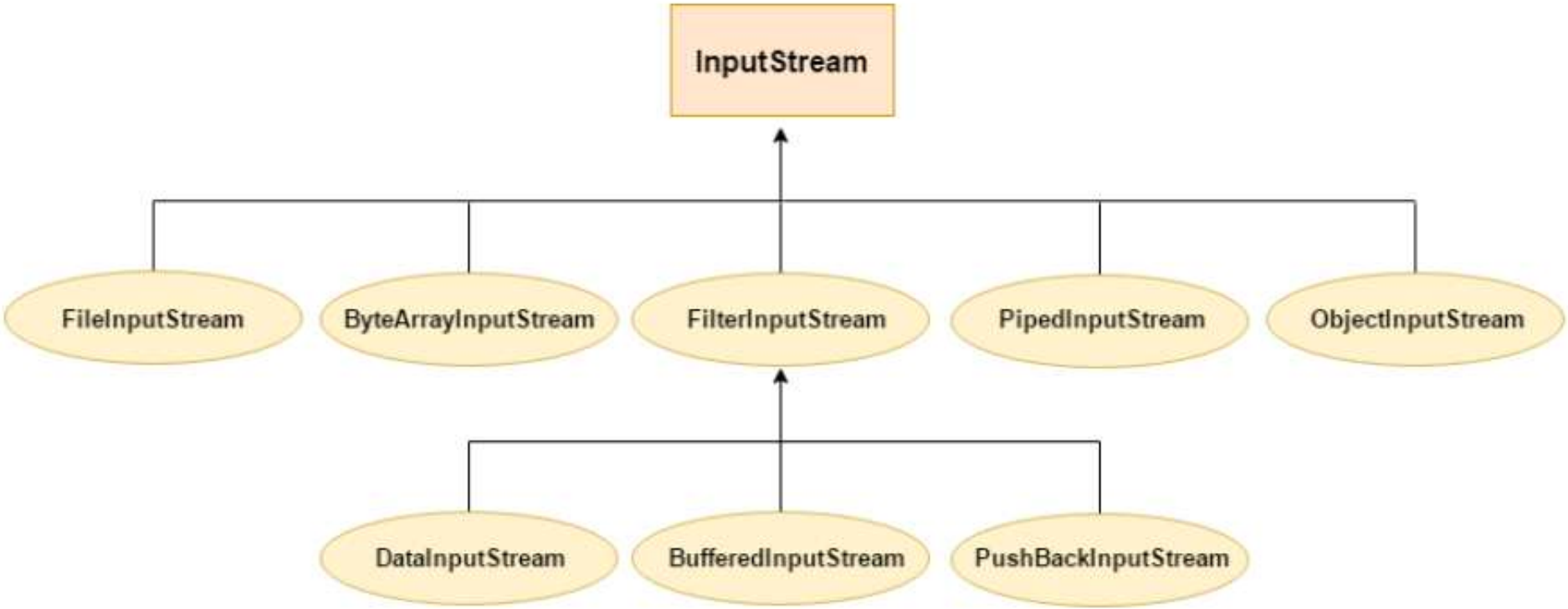
- An object from which we can read a sequence of bytes is called an ***input stream***.
- An object to which we can write a sequence of bytes is called an ***output stream***.
- An **input stream is a source** of bytes.
- An **output stream is a destination** for bytes.
- **Java I/O** (Input and Output) 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.

IO Stream

- input stream to read data from a source
- output stream to write data to a destination
- Source / Destination
  - file, an array, peripheral device or socket.



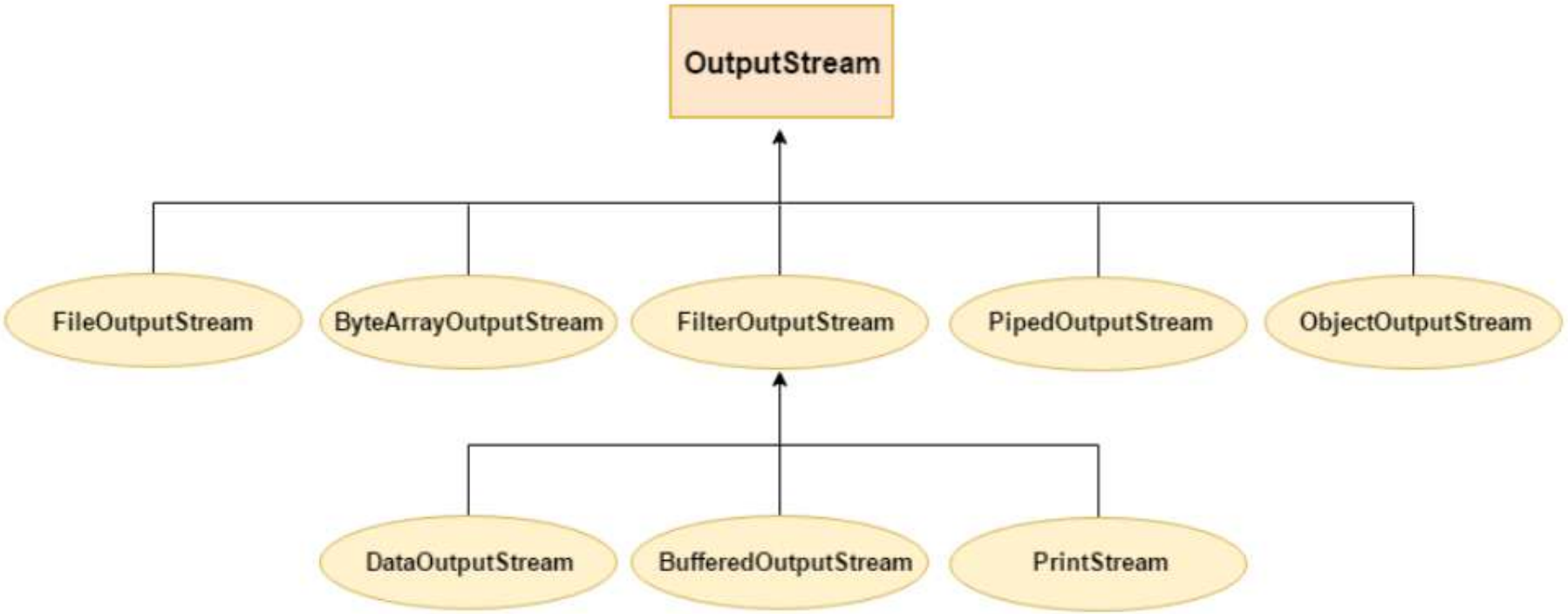
InputStream class



Useful methods of InputStream

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.

OutputStream Hierarchy



Useful methods of OutputStream

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.

Obtaining Streams

- **Easiest to use static methods**

```
Path path = Paths.get(filenameString); // We'll cover paths later this lesson
InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
```

- **Get an input stream from any URL:**

```
URL url = new URL("http://horstmann.com/index.html");
InputStream in = url.openStream();
```

- **Get an input stream from a byte[] array:**

```
byte[] bytes = ...;
InputStream in = new ByteArrayInputStream(bytes);
```

- **You can write to a ByteArrayOutputStream and then collect the bytes:**

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
Write to out
byte[] bytes = out.toByteArray();
```

Reading Bytes

- **The read method returns a single byte (as an int) or -1 at the end of input:**

```
InputStream in = ...;
int b = in.read();
if (b != -1) { byte value = (byte) b; ... }
```

- **It is more common to read bytes in bulk:**

```
byte[] bytes = ...;
int len = in.read(bytes);
```

- **No method for reading all bytes from a stream. Here is one solution:**

```
ByteArrayOutputStream out = new ByteArrayOutputStream();
byte[] bytes = new byte[1024];
while ((len = in.read(bytes)) != -1) out.write(bytes, 0, len);
bytes = out.toByteArray();
```

- **For files, just call:**

```
byte[] bytes = Files.readAllBytes(path);
```

Writing Bytes

- You can write one byte or bytes from an array:

```
OutputStream out = ...;
int b = ...;
out.write(b);
byte[] bytes = ...;
out.write(bytes);
out.write(bytes, start, length);
```

- When writing to a stream, close it when you are done:

```
out.close();
```

- Or better, use a try-with-resources block:

```
try (OutputStream out = ...) {
   out.write(bytes);
}
```

- To save an input stream to a file, call:

```
Files.copy(in, path, StandardCopyOption.REPLACE_EXISTING);
```

Read a character

- **int** i=System.in.read();
  - returns ASCII code of 1st character
- System.out.println((**char**)i);
  - will print the character

FileOutputStream Class

- Java **FileOutputStream** is an output stream used for writing data to a file.
- If you have to write primitive values into a file, use FileOutputStream class.
- You can write byte-oriented as well as character-oriented data through FileOutputStream class.
- But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

FileOutputStream class methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] a)	It is used to write a.length bytes from the byte array to the file output stream.
void write(byte[] a, 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.
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.

write byte to file

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

write string to file

```
import java.io.FileOutputStream;
public class FileOutputStreamExample
{
   public static void main(String args[]) {
        try{
          FileOutputStream fout=new FileOutputStream(“a.txt");
          String s="Welcome to www.BipinRupadiya.com";
          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

- 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.
- You can also read character-stream data.
- But, for reading streams of characters, it is recommended to use FileReader class.

Methods of FileInputStream class

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.
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.

read single character from file

```
import java.io.FileInputStream;
public class DataStreamExample  {
    public static void main(String args[]) {
        try {
            FileInputStream fin=new FileInputStream(“a.txt");
            int i=fin.read();
            System.out.print((char)i);
            fin.close();
        } catch(Exception e){System.out.println(e);}
    }
}
```

read all characters from file

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

BufferedOutputStream

- 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.
- For adding the buffer in an OutputStream, use the BufferedOutputStream class.
  - public class BufferedOutputStream extends FilterOutputStream { }
  - OutputStream os= new BufferedOutputStream(new FileOutputStream("a.txt"));

BufferedOutputStream

Constructor	Description
BufferedOutputStream(OutputStream os)	It creates the new buffered output stream which is used for writing the data to the specified output stream.
BufferedOutputStream(OutputStream os, int size)	It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

BufferedOutputStream

Method	Description
void write(int b)	It writes the specified byte to the buffered output stream.
void write(byte[] b, int off, int len)	It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
void flush()	It flushes the buffered output stream.

Let's take an example,

- we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object.
- The flush() flushes the data of one stream and send it into another.
- It is required if you have connected the one stream with another.

Example

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

BufferedInputStream

- Java **BufferedInputStream** class is used to read information from stream.
- It internally uses buffer mechanism to make the performance fast.
- **Important points:**
  - When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
  - When a BufferedInputStream is created, an internal buffer array is created.

BufferedInputStream class

```
Import java.io.*;
public class BufferedInputStream extends FilterInputStream
```

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the **BufferedInputStream** and saves it argument, the input stream IS, for later use.
BufferedInputStream(InputStream IS, int size)	It creates the **BufferedInputStream** with a specified buffer size and saves it argument, the input stream IS, for later use.

BufferedInputStream methods

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It read the next byte of data from the input stream.
int read(byte[] b, int off, int ln)	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.
void mark(int readlimit)	It sees the general contract of the mark method for the input stream.
long skip(long x)	It skips over and discards x bytes of data from the input stream.
boolean markSupported()	It tests for the input stream to support the mark and reset methods.

Example

```
import java.io.*;
public class BufferedInputStreamExample  {
        public static void main(String args[])  {
            try {
                        FileInputStream fin=new FileInputStream("a.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

- **SequenceInputStream** is used to read data from multiple streams.
- It reads data sequentially (one by one).
- **public class SequenceInputStream extends InputStream**

Constructor	Description
SequenceInputStream(InputStream s1, InputStream s2)	creates a new input stream by reading the data of two input stream in order, first s1 and then s2.
SequenceInputStream(Enumeration e)	creates a new input stream by reading the data of an enumeration whose type is InputStream.

SequenceInputStream

Method	Description
int read()	It is used to read the next byte of data from the input stream.
int read(byte[] ary, int off, int len)	It is used to read len bytes of data from the input stream into the array of bytes.
int available()	It is used to return the maximum number of byte that can be read from an input stream.
void close()	It is used to close the input stream.

SequenceInputStream Example-1

```
import java.io.*;
class SequenceInputStreamExample1 {
        public static void main(String args[])throws Exception    {
            FileInputStream i1=new FileInputStream("a.txt");
            FileInputStream i2=new FileInputStream("b.txt");
            SequenceInputStream inst=new SequenceInputStream(i1, i2);
            int j;
            while((j=inst.read())!=-1)  {
                        System.out.print((char)j);
            }
            inst.close();
            i1.close();
            i2.close();
        }
}
```

Example-2
Read from two file & write it to one file

```
import java.io.*;
class   SequenceInputStreamExample2 {
{
        public static void main(String args[])throws Exception   {
                FileInputStream fin1=new FileInputStream("a.txt");
                FileInputStream fin2=new FileInputStream("b.txt");
                FileOutputStream fout=new FileOutputStream("c.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

- **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 buffer of **ByteArrayOutputStream** automatically grows according to data.
- **public class** ByteArrayOutputStream **extends** OutputStream

Constructor	Description
ByteArrayOutputStream()	Creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
ByteArrayOutputStream(int size)	Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

ByteArrayOutputStream class methods

Method	Description
int size()	returns the current size of a buffer.
byte[] toByteArray()	create a newly allocated byte array.
String toString()	for converting the content into a string decoding bytes using a platform default character set.
String toString(String charsetName)	for converting the content into a string decoding bytes using a specified charsetName.
void write(int b)	for writing the byte specified to the byte array output stream.
void write(byte[] b, int off, int len	for writing len bytes from specified byte array starting from the offset off to the byte array output stream.
void writeTo(OutputStream out)	for writing the complete content of a byte array output stream to the specified output stream.
void reset()	to reset the count field of a byte array output stream to zero value.
void close()	to close the ByteArrayOutputStream.

ByteArrayOutputStream Example

```
import java.io.*;
public class ByteArrayOutputStreamExample {
      public static void main(String args[])throws Exception {
            FileOutputStream fout1=new FileOutputStream("a.txt");
            FileOutputStream fout2=new FileOutputStream("b.txt");
            ByteArrayOutputStream bout=new ByteArrayOutputStream();
            bout.write(65);
            bout.writeTo(fout1);
            bout.writeTo(fout2);
            bout.flush();
            bout.close();
            System.out.println("Success...");
      }
}
```

ByteArrayInputStream

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

Constructor	Description
ByteArrayInputStream(byte[] a)	Creates a new byte array input stream which uses array as its buffer array.
ByteArrayInputStream(byte[] ary, int offset, int len)	Creates a new byte array input stream which uses array as its buffer array that can read up to specified len bytes of data from an array.

ByteArrayInputStream class methods

Methods	Description
int available()	return the number of remaining bytes that can be read from the input stream.
int read()	read the next byte of data from the input stream.
int read(byte[] ary, int off, int len)	read up to len bytes of data from an array of bytes in the input stream.
boolean markSupported()	test the input stream for mark and reset method.
long skip(long x)	skip the x bytes of input from the input stream.
void mark(int readAheadLimit)	set the current marked position in the stream.
void reset()	reset the buffer of a byte array.
void close()	for closing a ByteArrayInputStream.

Example

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

- It 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.
- **public class** DataOutputStream
  extends FilterOutputStream
  implements DataOutput

DataOutputStream class methods

Method	Description
int size()	return the number of bytes written to the data output stream.
void write(int b)	write the specified byte to the underlying output stream.
void write(byte[] b, int off, int len)	write len bytes of data to the output stream.
void writeBoolean(boolean v)	write Boolean to the output stream as a 1-byte value.
void writeChar(int v)	write char to the output stream as a 2-byte value.
void writeChars(String s)	write String to the output stream as a sequence of characters.
void writeByte(int v)	write a byte to the output stream as a 1-byte value.
void writeBytes(String s)	write string to the output stream as a sequence of bytes.
void writeInt(int v)	write an int to the output stream
void writeShort(int v)	write a short to the output stream.
void writeLong(long v)	write a long to the output stream.
void writeUTF(String str)	write a string to the output stream using UTF-8 encoding
void flush()	flushes the data output stream.

DataInputStream

- It allows an application to read primitive data from the input 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.
- **public class** DataInputStream **extends** FilterInputStream **implements** DataInput

DataInputStream class Methods

Method	Description
int read(byte[] b)	read the number of bytes from the input stream.
int read(byte[] b, int off, int len)	read len bytes of data from the input stream.
int readInt()	read input bytes and return an int value.
byte readByte()	read and return the one input byte.
char readChar()	read two input bytes and returns a char value.
double readDouble()	read eight input bytes and returns a double value.
boolean readBoolean()	read one input byte and return true if byte is non zero, false if byte is zero.
int skipBytes(int x)	skip over x bytes of data from the input stream.
String readUTF()	read a string that has been encoded using the UTF-8 format.
void readFully(byte[] b)	read bytes from the input stream and store them into the buffer array.
void readFully(byte[] b, int off, int len)	It is used to read len bytes from the input stream.

Example

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

FilterOutputStream Class

- It implements the OutputStream class.
- It provides different sub classes such as BufferedOutputStream and DataOutputStream to provide additional functionality.
- So it is less used individually.

Method	Description
void write(int b)	write the specified byte to the output stream.
void write(byte[] a)	write a.length byte to the output stream.
void write(byte[] b, int off, int len)	write len bytes from the offset off to the output stream.
void flush()	flushes the output stream.
void close()	close the output stream.

Example

```
import java.io.*;
public class FilterExample {
        public static void main(String[] args) throws IOException {
                File data = new File("a.txt");
                FileOutputStream file = new FileOutputStream(data);
                FilterOutputStream filter = new FilterOutputStream(file);
                String s="Welcome to JVIMS";
                byte b[]=s.getBytes();
                filter.write(b);
                filter.flush();
                filter.close();
                file.close();
                System.out.println("Success...");
        }
}
```

FilterInputStream

- It implements the InputStream.
- It contains different sub classes as BufferedInputStream, DataInputStream for providing additional functionality.
- So it is less used individually.

Method	Description
int available()	return an estimate number of bytes that can be read from the input stream.
int read()	read the next byte of data from the input stream.
int read(byte[] b)	read up to byte.length bytes of data from the input stream.
long skip(long n)	skip over and discards n bytes of data from the input stream.
boolean markSupported()	test if the input stream support mark and reset method.
void mark(int readlimit)	mark the current position in the input stream.
void reset()	reset the input stream.
void close()	close the input stream.

Example

```
import java.io.*;
public class FilterInputStreamExample {
        public static void main(String[] args) throws IOException {
            File data = new File("a.txt");
            FileInputStream  file = new FileInputStream(data);
            FilterInputStream filter = new BufferedInputStream(file);
            int k =0;
            while((k=filter.read())!=-1){
               System.out.print((char)k);
            }
            file.close();
            filter.close();
        }
}
```

RandomAccessFile

- This class is used for reading and writing to random access file.
- A random-access file has a *file pointer* that indicates the position of the next byte to be read or written.
- If end-of-file is reached before the desired number of byte has been read than EOFException is thrown.
- It is a type of IOException.
- You can open a random-access file either for reading only or for both reading and writing
  - RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
  - RandomAccessFile inOut = new RandomAccessFile("employee.dat", "rw");

RandomAccessFile methods

Method	Method
void close()	closes this random access file stream and releases any system resources associated with the stream.
Int readInt()	reads a signed 32-bit integer from this file.
String readUTF()	reads in a string from this file.
void seek(long pos)	sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void writeDouble(double v)	converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
void writeFloat(float v)	converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
void write(int b)	writes the specified byte to this file.
int read()	reads a byte of data from this file.
long length()	returns the length of this file.
void seek(long pos)	sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

RandomAccessFile constructors

Constructor	Description
RandomAccessFile(File file, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Example

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample
{
        static final String FILEPATH ="a.txt";
        public static void main(String[] args)
        {
            try
            {
                System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
                writeToFile(FILEPATH, "I love my country and my people", 31);
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
        }
```

Conti…

```
private static byte[] readFromFile(String filePath, int position, int size) throws
IOException  {
        RandomAccessFile file = new RandomAccessFile(filePath, "r");
        file.seek(position);
        byte[] bytes = new byte[size];
        file.read(bytes);
        file.close();
        return bytes;
}

private static void writeToFile(String filePath, String data, int position) throws
IOException  {
        RandomAccessFile file = new RandomAccessFile(filePath, "rw");
        file.seek(position);
        file.write(data.getBytes());
        file.close();
    }
}
```

FileWriter

- Java FileWriter class is used to write character-oriented data to a file.
- It is character-oriented class which is used for file handling in java.
- Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.
- public class FileWriter extends OutputStreamWriter

FileWriter

Constructor	Description
FileWriter(String file)	Creates a new file. It gets file name in string.
FileWriter(File file)	Creates a new file. It gets file name in File object.

Method	Description
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void write(char[] c)	It is used to write char array into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

FileWriter Example

```
import java.io.FileWriter;
public class FileWriterExample
{
   public static void main(String args[])
   {
       try{
                FileWriter fw=new FileWriter("D:\\a.txt");
                fw.write("Welcome to JVIMS.");
                fw.close();
        }
         catch(Exception e){System.out.println(e);}
                System.out.println("Success...");
    }
}
```

FileReader

- Java FileReader class is used to read data from the file.
- It returns data in byte format like FileInputStream class.
- It is character-oriented class which is used for file handling in java.
- **public class** FileReader **extends** InputStreamReader

FileReader

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Method	Description
int read()	It is used to return a character in ASCII form. It returns -1 at the end of file.
void close()	It is used to close the FileReader class.

Example

```
import java.io.FileReader;
public class FileReaderExample
{
    public static void main(String args[]) throws Exception
    {
        FileReader fr=new FileReader("D:\\a.txt");
        int i;
        while((i=fr.read())!=-1)
            System.out.print((char)i);
        fr.close();
    }
}
```

Path

- A Path is a sequence of directory names, optionally followed by a file name.
- The first component of a path may be a root component such as **/** or **C:**.
- The permissible root components depend on the file system.
- A path that starts with a root component is absolute. Otherwise, it is relative.
- For the absolute path, we assume a UNIX-like file system.
  - Path absolute = Paths.get("/home", "harry");
  - Path relative = Paths.get("myprog", "conf", "user.properties");

Paths.get()

- The static **Paths.get()** receives one or more strings, which it joins with the path separator of the default file system (/ for a UNIX-like file system, \ for Windows).
- It then parses the result, throwing an InvalidPathException if the result is not a valid path in the given file system.
- The result is a Path object.

Example (UNIX)

- Path p = Paths.get("/home", "fred", "myprog.properties");
- Path parent = p.getParent(); // the path /home/fred
- Path file = p.getFileName(); // the path myprog.properties
- Path root = p.getRoot(); // the path /

Example (windows)

```
import java.nio.file.Path;
import java.nio.file.Paths;
public class PathExample
{
    public static void main(String[] args)
   {
                Path path = Paths.get("c:\\data\\myfile.txt");
                System.out.println("path = " + path);
                Path currentDir = Paths.get(".");
                System.out.println(currentDir.toAbsolutePath());
                Path path2 = currentDir.toAbsolutePath().normalize();
                System.out.println("path2 = " + path2);
    }
}
```

File Class

- The File class is an abstract representation of file and directory pathname.
- A pathname can be either absolute or relative.
- The File class have several methods for working with directories and files such as
  - creating new directories or files,
  - deleting and renaming directories or files,
  - listing the contents of a directory etc.

File Constructor

Constructor	Description
File(File parent, String child)	It creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname)	It creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child)	It creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri)	It creates a new File instance by converting the given file: URI into an abstract pathname.

File's Methods

Method	Description
boolean createNewFile()	It atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean canWrite()	It tests whether the application can modify the file denoted by this abstract pathname.String[]
boolean canExecute()	It tests whether the application can execute the file denoted by this abstract pathname.
boolean canRead()	It tests whether the application can read the file denoted by this abstract pathname.
boolean isAbsolute()	It tests whether this abstract pathname is absolute.
boolean isDirectory()	It tests whether the file denoted by this abstract pathname is a directory.
boolean isFile()	It tests whether the file denoted by this abstract pathname is a normal file.
String getName()	It returns the name of the file or directory denoted by this abstract pathname.
String getParent()	It returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
Path toPath()	It returns a java.nio.file.Path object constructed from the this abstract path.
URI toURI()	It constructs a file: URI that represents this abstract pathname.
File[] listFiles()	It returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname
long getFreeSpace()	It returns the number of unallocated bytes in the partition named by this abstract path name.
String[] list(FilenameFilter filter)	It returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
Boolean mkdir()	It creates the directory named by this abstract pathname.

Example

```
import java.io.*;
public class FileDemo {
   public static void main(String[] args) {

      try {
         File file = new File("D:\\FileDemo.txt");
         if (file.createNewFile()) {
            System.out.println("New File is created!");
         } else {
            System.out.println("File already exists.");
         }
      }
                              catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Example

List file from current directory

```
import java.io.*;
public class FileListExample {
        public static void main(String[] args) {
                File dir=new File(".");
                File files[]=dir.listFiles();
                for(File file:files)
                {
                        System.out.println("\n"+file.getName()+
                        "\nCan Write: "+file.canWrite()+
                        " Is Hidden: "+file.isHidden()+
                        " Length: "+file.length()+" bytes");
                }
        }
}
```

Reading and Writing Files

- The Files class makes quick work of common file operations.
- For example, you
- **can easily read the entire contents of a file:**
  - byte[] bytes = Files.readAllBytes(path);
- **If you want to read the file as a string, call readAllBytes followed by**
  - String content = new String(bytes, charset);
- **But if you want the file as a sequence of lines, call**
  - List<String> lines = Files.readAllLines(path, charset);
- **if you want to write a string, call**
  - String content =“I Love JVIMS”
  - Files.write(path, content.getBytes(charset));
- **To append to a given file, use**
  - Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);

Reading and Writing Files

- **You can also write a collection of lines with**
  - Files.write(path, lines);
- **These simple methods are intended for dealing with text files of moderate length.**
- **If your files are large or binary, you can still use the familiar input/output streams or readers/writers:**
  - InputStream in = Files.newInputStream(path);
  - OutputStream out = Files.newOutputStream(path);
  - Reader in = Files.newBufferedReader(path, charset);
  - Writer out = Files.newBufferedWriter(path, charset);
- These methods save you from dealing with FileInputStream, FileOutputStream, BufferedReader, or BufferedWriter.

Creating Files and Directories

- **To create a new directory, call**
  - Files.createDirectory(path);
  - All but the last component in the path must already exist.
- **To create intermediate directories as well, use**
  - Files.createDirectories(path);
- **You can create an empty file with**
  - Files.createFile(path);
  - The call throws an exception if the file already exists.
- **There are methods for creating a temporary file or directory in a given or system-specific location.**
  - Path newPath = Files.createTempFile(dir, prefix, suffix);
  - Path newPath = Files.createTempFile(prefix, suffix);
  - Path newPath = Files.createTempDirectory(dir, prefix);
  - Path newPath = Files.createTempDirectory(prefix);

Copying, Moving Files

- **To copy a file from one location to another, simply call**
  - Files.copy(fromPath, toPath);
- **To move the file (that is, copy and delete the original), call**
  - Files.move(fromPath, toPath);
- The copy or move will fail if the target exists.
- If you want to overwrite an existing target, use the REPLACE_EXISTING option.
- If you want to copy all file attributes, use the COPY_ATTRIBUTES option.
- **You can supply both like this:**
  - Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.COPY_ATTRIBUTES);

Copying, Moving Files

- **The copy or move will fail if the target exists.**
- **If you want to overwrite an existing target,**
  - use the REPLACE_EXISTING option.
- **If you want to copy all file attributes,**
  - use the COPY_ATTRIBUTES option.
  - You can supply both like this:
  - Files.copy(fromPath, toPath, StandardCopyOption.REPLACE_EXISTING, StandardCopyOption.COPY_ATTRIBUTES);
- **You can specify that a move should be atomic.**
- **Then you are assured that either the move completed successfully, or the source continues to be present.**
- **Use the ATOMIC_MOVE option:**
  - Files.move(fromPath, toPath, StandardCopyOption.ATOMIC_MOVE);

Copying, Moving Files

- You can also copy an input stream to a Path, which just means saving the input stream to disk.
- Similarly, you can copy a Path to an output stream.
- Use the following calls:
  - Files.copy(inputStream, toPath);
  - Files.copy(fromPath, outputStream);

Deleting Files

- Finally, to delete a file, simply call
  - Files.delete(path);
- This method throws an exception if the file doesn't exist.
- instead you should use
  - boolean deleted = Files.deleteIfExists(path);

Summery of copy move & delete

- static Path copy(Path from, Path to, CopyOption... options)
- static Path move(Path from, Path to, CopyOption... options)

- static long copy(InputStream from, Path to, CopyOption... options)
- static long copy(Path from, OutputStream to, CopyOption... options)

- static void delete(Path path)
- static boolean deleteIfExists(Path path)

Getting File Information

- The following static methods return a boolean value to check a property of a path:
  - exists()
  - isHidden()
  - isReadable(), isWritable(), isExecutable()
  - isRegularFile(), isDirectory(), isSymbolicLink()
- The size method returns the number of bytes in a file.
  - long fileSize = Files.size(path);
- The getOwner() returns the owner of the file, as an instance of
  - java.nio.file.attribute.UserPrincipal.

BasicFileAttributes

- All file systems report a set of basic attributes, encapsulated by the **BasicFileAttributes** interface,
- **The basic file attributes are**
  - The times at which the file was created, last accessed, and last modified, as instances of the class java.nio.file.attribute.FileTime
  - Whether the file is a regular file, a directory, a symbolic link, or none of these
  - file size
- BasicFileAttributes attributes = Files.readAttributes(path, BasicFileAttributes.class);

Methods of java.nio.file.attribute.BasicFileAttributes

- FileTime creationTime()
- FileTime lastAccessTime()
- FileTime lastModifiedTime()
- boolean isRegularFile()
- boolean isDirectory()
- boolean isSymbolicLink()
- long size()

Contact:

Bipin S. Rupadiya

Assistant Professor, JVIMS

Mo. : +91-9228582425

Email: info@bipinrupadiya.com

Blog : www.BipinRupadiya.com