File

File

- Although most of the classes defined by **java.io** operate on streams, the **File** class does not.

- It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself.

- A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

Constructors of **File**

- File(String *directoryPath*)
- File(String *directoryPath*, String *filename*)
- File(File *dirObj*, String *filename*)

- Example:
  - File f1 = new File(“C:/autoexec.bat");
  - File f2 = new File(“C:/", "autoexec.bat");

  - File f3 = new File(“C:/”);
  - File f4 = new File(f3, "autoexec.bat");

File methods

```
import java.io.*;
public class Test
{
    public static void main(String args[])
    {
        File f = new File("C:/Users/hp/Desktop/abc.txt");

        System.out.println("File name: " + f.getName());
        System.out.println("File path: " + f.getPath());
        System.out.println("Is it absolute: " + f.isAbsolute());
        System.out.println("File absolute-path: " + f.getAbsolutePath());
        System.out.println("File exists: " + f.exists());
        System.out.println("Is it readable: " + f.canRead());
        System.out.println("Is it writeable: " + f.canWrite());
        System.out.println("Is it a file: " + f.isFile());
        System.out.println("Is it a directory: " + f.isDirectory());
        System.out.println("File last-modified: " + f.lastModified());
        System.out.println("File size: " + f.length());
        //System.out.println("Deleting file: " + f.delete());
        System.out.println("Rename file: " + f.renameTo(new File("C:/Users/hp/Desktop/def.txt")));
    }
}
```

Basics of I/O Streams

Introduction

- Most real applications of Java are not text-based, console programs.
- Java’s support for console I/O is limited.
- Text-based console I/O is not very important to Java programming.
- Java does provide strong, flexible support for I/O as it relates to files and networks.

Streams

- Java implements streams within class hierarchies defined in the **java.io** package.
- A stream is a sequence of data.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
- An I/O Stream represents an input source or an output destination.

I/O Streams

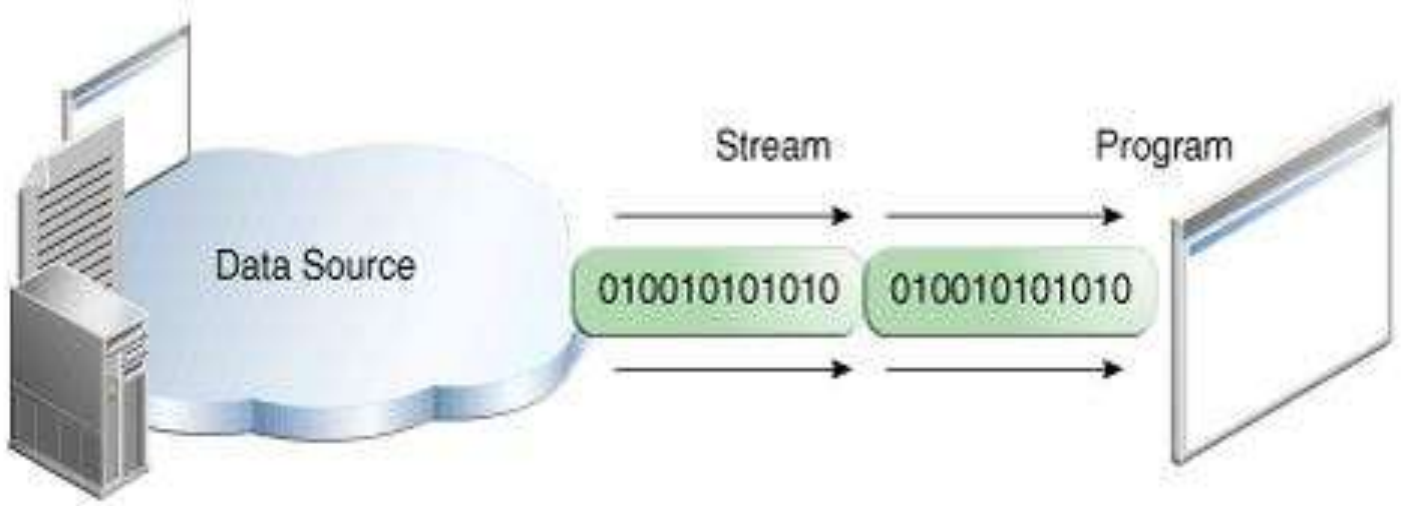
- A stream can represent many different kinds of sources and destinations
  – disk files, devices, other programs, a network socket, and memory arrays

- Streams support many different kinds of data
  – simple bytes, primitive data types, localized characters, and objects

- Some streams simply pass on data; others manipulate and transform the data in useful ways.

Input Stream

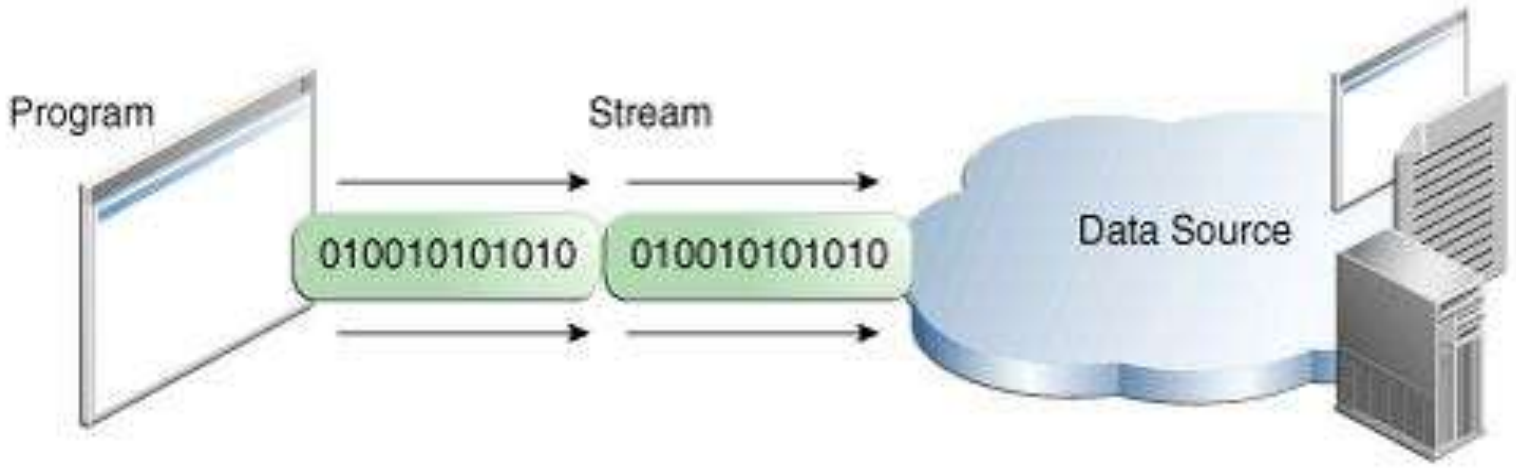
- A program uses an input stream to read data from a source, one item at a time.



Reading information into a program.

Output Stream

- A program uses an *output stream* to write data to a destination, one item at time:



Writing information from a program.

Types of Streams

- Java defines two different types of Streams-
  - ***Byte Streams***
  - ***Character Streams***

- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.
- Character streams provide a convenient means for handling input and output of characters.
- In some cases, *character streams are more efficient than byte streams*.

Key points

- In Java, a byte is not the same thing as a char . Therefore a **byte stream** is different from a **character stream**. So, Java defines two types of streams: Byte Streams and Character Streams .

- A **byte stream** access the file byte by byte. Java programs use byte streams to perform input and output of 8-bit bytes. It is suitable for any kind of file, however not quite appropriate for text files. For example, if the file is using a unicode encoding and a character is represented with two bytes, the byte stream will treat these separately and you will need to do the conversion yourself.

- Byte oriented streams do not use any encoding scheme while Character oriented streams use character encoding scheme(UNICODE). All byte stream classes are descended from InputStream and OutputStream .

Byte Streams

- Programs use *byte streams* to perform input and output of 8-bit bytes.
- Byte streams are defined by using two class hierarchies.
- At the top, there are two abstract classes: **InputStream** and **OutputStream**.
- The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement.
- Two of the most important methods are **read( )**and **write( )**, which, respectively, read and write bytes of data.
- Both methods are declared as abstract inside InputStream and OutputStream.

Byte Stream Contd…

Note: read() returns an int value.

- Using a int as a return type allows read() to use -1 to indicate that it has reached the end of the stream.

Example 1 (reading file using byte-stream)

```
import java.io.*;
class Test {
    public static void main(String args[]) throws IOException
    {
        FileInputStream fp = null;
        try {
            fp = new FileInputStream("C:/Users/hp/Desktop/abc.txt");
            int c = 0 ;
            while( c != -1 )    //till end of file is not reached
            {
                c = fp.read();  //reading 1-byte at a time
                if(c==-1)
                    break;

                System.out.print((char)c);
            }
        }
        catch(IOException e) {
            System.out.println("Error occured while reading file...");
        }
        finally {
            if(fp!=null)
                fp.close();
        }
    }
}
```

FileInputStream

Its two most common constructors are shown here:

FileInputStream(String *filepath*)
FileInputStream(File *fileObj*)

The following example creates two **FileInputStreams** that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Example 2 (reading file using byte-stream)

```
import java.io.*;
class Test {
    public static void main(String args[]) throws IOException
    {
        FileInputStream fp = null;
        try {
            fp = new FileInputStream("C:/Users/hp/Desktop/abc.txt");
            int c;
            while( (c = fp.read()) != -1 )
            {
                System.out.print((char)c);
            }
        }
        catch(IOException e) {
            System.out.println("Error occured while reading file...");
        }
        finally {
            try {
                if(fp!=null)
                    fp.close();
            }
            catch(IOException e) {
                System.out.println("Error while closing file...");
            }
        }
    }
}
```

Closing the Streams

- Closing a stream when it's no longer needed is very important.
- It is so important that we have used a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.
- One possible error is that ***Test*** *class* (in previous example) was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value.
- That's why ***Test*** *class* (in previous example) makes sure that each stream variable contains an object reference before invoking close.

try with resources

- A resource is an object that must be closed once your program is done using it. For example a File resource or JDBC resource for database connection or a Socket connection resource.

- Before Java 7, there was no auto resource management and we should explicitly close the resource once our work is done with it. Usually, it was done in the finally block of a try-catch statement.

- This approach used to cause memory leaks and performance hit when we forgot to close the resource.

- **Some of the benefits of using try with resources in java are**:
  - More readable code and easy to write.
  - Automatic resource management.
  - Number of lines of code is reduced.
  - No need of finally block just to close the resources.
  - We can open multiple resources in try-with-resources statement separated by a semicolon.

Example 1

```
import java.io.*;
class Test {
    public static void main(String args[]) throws IOException
    {
        try (FileInputStream fp = new FileInputStream("C:/Users/hp/Desktop/abc.txt"))
        {
            int c = 0;
            while( c!= -1)  {
                c = fp.read();
                if(c== -1)
                    break;
                System.out.print((char)c);
            }
        }
        catch(IOException e) {
            System.out.println("Error while reading file...");
        }
    }
}
```

Example 2

Before Java 9 a resource that is to be automatically closed must be created inside the parentheses of the *try* block of a try-with-resources construct. From Java 9, this is no longer necessary. If the variable referencing the resource is effectively final, you can simply enter a reference to the variable inside the try block parentheses. Here is an example of the Java 9 *try-with-resources* enhancement:

```
private static void printFile() throws IOException {
    FileInputStream input = new FileInputStream("file.txt");
    try(input) {

        int data = input.read();
        while(data != -1){
            System.out.print((char) data);
            data = input.read();
        }
    }
}
```

Methods defined by 'InputStream'

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

FileInputStream

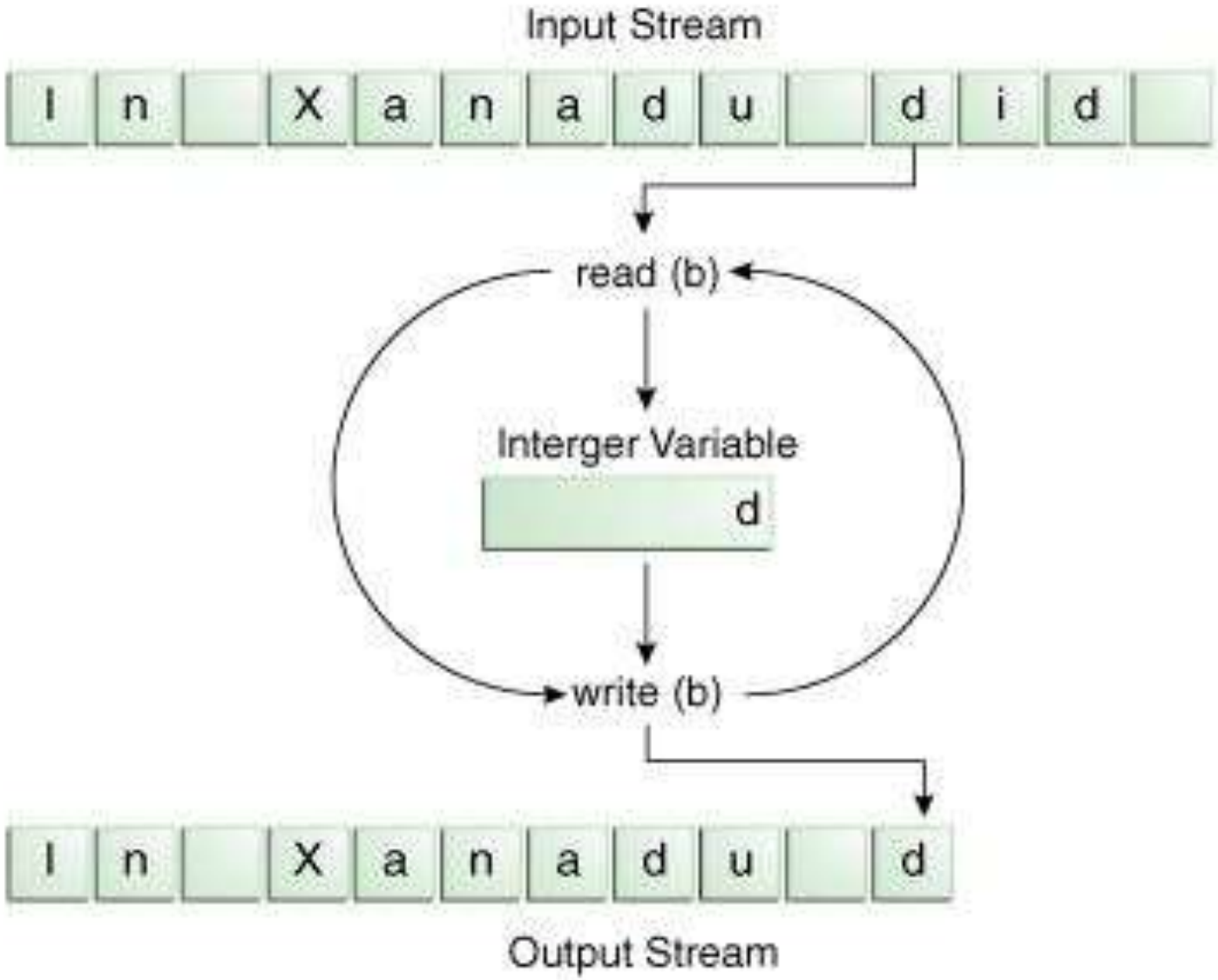
```
FileInputStream fin = new FileInputStream("C:/Users/hp/Desktop/temp/copy.txt");
byte bin[] = new byte[100];
int c;

while( (c=fin.read() )!=-1) //Reading 1-char at-a-time
{
    System.out.print( (char)c );
}

/*
if( (c= fin.read(bin) ) !=-1)   //Reading bytes in whole array
    System.out.println(new String(bin,0,bin.length));
else
    System.out.println("Could not read");       */


/*
if( (c= fin.read(bin,4,20) )!=-1 )  //Reading 20 bytes into array starting from 4th index
    System.out.println(new String(bin,0,b.length));
else
    System.out.println("Could not read");       */
```

Reading/ Writing File using Streams



FileOutputStream

- Used to write bytes to file.
- Commonly used constructors:

  FileOutputStream (**String** *filePath*)

  FileOutputStream (**File** *fileObj*)

  FileOutputStream (**String** *filePath,* **boolean** *append*)

  FileOutputStream (**File** *fileObj,* **boolean** *append*)

Example (copying file using byte-stream)

```
import java.io.*;
class Test {
    public static void main(String args[]) throws IOException  {
        FileInputStream fin = null;
        FileOutputStream fout = null;
        try {
            fin = new FileInputStream("C:/Users/hp/Desktop/abc.txt");
            fout = new FileOutputStream("C:/Users/hp/Desktop/result.txt");
            int c = 0;
            while( c!= -1 ) {
                c = fin.read();
                if(c==-1)
                    break;
                fout.write(c);
            }
        }
        catch(IOException e) {
            System.out.println("Error while reading or writing file....");
        }
        finally {
            if(fin!=null)
                fin.close();
            if(fout!=null)
                fout.close();
        }
    }
}
```

Methods defined by ‘OutputStream’

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

FileOutputStream

- Used to write bytes to file.
- Commonly used constructors:

  FileOutputStream (**String** *filePath*)

  FileOutputStream (**File** *fileObj*)

  FileOutputStream (**String** *filePath,* **boolean** *append*)

  FileOutputStream (**File** *fileObj,* **boolean** *append*)

```
FileOutputStream fout = new FileOutputStream("C:/Users/hp/Desktop/temp/copy.txt");
String str = new String("The file is going to be written with this string now");
byte b[] = str.getBytes();

for(int i=0; i<b.length; i++)
{
    fout.write(b[i]);       //Writing 1-byte at-a-time
}

// fout.write(b);           //Writing whole byte-array at-a-time
// fout.write(b, 0, 20);    //Writing 20 bytes from byte-array from 0th index
```

Exercise 1

Write a program to read the contents of a file whose location is taken using command-line argument. Verify whether path/file exists before reading & displaying its contents on the console screen. Use proper exception handling in your program.

Exercise 2

Write a program to take the input from the user (in form of string). Take the location & file name using command-line arguments. If the file-name already exists, then keep on asking the user to enter new file-name (until he enters non-existing file).

Now write the input (taken from the user) in the file and print how many bytes have been written onto the file.

Use proper exception handling in your program.

Exercise 3

1) Write a program to read the contents of a file whose location is taken using command-line argument. Verify whether path exists or not. If path given in incorrect, then it should keep on asking the correct path from the user.

2) If path entered is of the directory, then it should ask the name of the file to read and keep on verifying its existence again until correct file is mentioned.

3) Print the following information about the file: size, readable/writable.

4) Read the contents of the file and display them on the console screen. Use proper exception handling in your program.

Use of Byte Stream

- Byte streams should only be used for the most primitive I/O.
- Since a file contains character data, the best approach is to use character streams.
- Byte Stream represents a kind of low-level I/O.
- So why talk about byte streams?
- Because all other stream types are built on byte streams.

Byte Stream Classes

Stream Class	Meaning / Use
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
DataInputStream	contains methods for reading the Java standard data types
DataOutputStream	contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PrintStream	Output stream that contains print() and println( )
PipedInputStream	Input Pipe
PipedOutputStream	Output Pipe

Character Stream

- The Java platform stores character values using Unicode conventions.
- Character stream I/O automatically translates this internal format to and from the local character set.
- Character streams are defined by using two class hierarchies.
- At the top are two abstract classes, **Reader** and **Writer**.
- These abstract classes handle Unicode character streams.
- Similar to Byte Streams, **read()** and **write()** methods are defined in **Reader** and **Writer** class.

Character Stream Classes

Stream Class	Meaning
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Methods defined by 'Reader'

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported on this stream.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.

Methods defined by 'Writer'

Method	Description
Writer append(char *ch*)	Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*)	Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream.
Writer append(CharSequence *chars*, int *begin*, int *end*)	Appends the subrange of *chars* specified by *begin* and *end*–1 to the end of the invoking ouput stream. Returns a reference to the invoking stream.
abstract void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with expressions without having to cast them back to **char**.

Predefined Streams

- **java.lang** package defines a class called *System*, which encapsulates several aspects of the run-time environment.
- System contains three predefined stream variables:

  in, *out*, and *err*.
- These fields are declared as public, static, and final within System.
- This means that they can be used by any other part of your program and without reference to a specific System object.

Predefined Streams

- *System.out* refers to the standard output stream. By default, this is the console.
- *System.in* refers to standard input, which is the keyboard by default.
- *System.err* refers to the standard error stream, which also is the console by default.
- However, these streams may be redirected to any compatible I/O device.

- *System.in* is an object of type *InputStream*;
- *System.out* and *System.err* are objects of type *PrintStream*.

- These are *byte streams*, even though they typically are used to read and write characters from and to the console.

FileWriter

```java
import java.io.*;
class Test {
    public static void main(String args[]) throws IOException
    {
        FileWriter fw = null;
        try {
            fw = new FileWriter("C:/Users/hp/Desktop/result.txt");
            String str = "This is going to be written in file.";
            fw.write(str);
            /*
                char c[] = new char[str.length()];
                str.getChars(0, str.length(), c, 0);
                for(int i=0; i<c.length; i++)
                fw.write(c[i]);          //writing 1 character at a time
            */
        }
        catch(IOException e) {
            System.out.println("Error while writing the file...");
        }
        finally {
            fw.close();
        }
    }
}
```

- **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.

FileReader

```
import java.io.*;
class Test {
    public static void main(String args[]) throws IOException
    {
        FileReader fr = null;
        try {
            fr = new FileReader("C:/Users/hp/Desktop/result.txt");
            int c = 0;
            while(c!= -1) {
                c = fr.read();
                if(c == -1)
                    break;
                System.out.println((char)c);
            }
        }
        catch(IOException e) {
            System.out.println("Error while reading the file...");
        }
        finally {
            fr.close();
        }
    }
}
```

- **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.

PrintWriter

- **PrintWriter** class is the implementation of Writer class.
- It is used to print the formatted representation of objects to the text-output stream.

Methods of **PrintWriter**

java.io.PrintWriter	
`+PrintWriter(file: File)`	Creates a `PrintWriter` object for the specified file object.
`+PrintWriter(filename: String)`	Creates a `PrintWriter` object for the specified file-name string.
`+print(s: String): void`	Writes a string to the file.
`+print(c: char): void`	Writes a character to the file.
`+print(cArray: char[]): void`	Writes an array of characters to the file.
`+print(i: int): void`	Writes an `int` value to the file.
`+print(l: long): void`	Writes a `long` value to the file.
`+print(f: float): void`	Writes a `float` value to the file.
`+print(d: double): void`	Writes a `double` value to the file.
`+print(b: boolean): void`	Writes a `boolean` value to the file.
Also contains the overloaded `println` methods.	A `println` method acts like a `print` method; additionally, it prints a line separator. The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on Unix.
Also contains the overloaded `printf` methods.	The `printf` method was introduced in §3.16, "Formatting Console Output."

Methods of **PrintWriter**

PrintWriter append(char c)	It is used to append the specified character to the writer.
PrintWriter append(CharSequence ch)	It is used to append the specified character sequence to the writer.
void print(Object obj)	It is used to print an object.
void flush()	It is used to flushes the stream.

Example

```
import java.io.*;
class Test {
    public static void main(String args[]) throws IOException
    {
        PrintWriter pw = null;
        try {
            pw = new PrintWriter("C:/Users/hp/Desktop/result.txt");
            pw.println("It will move to next line after writing");
            pw.print("Marks: ");
            pw.println(28);
            pw.flush();
            char c[] = {'a','r','r','a','y'};
            pw.print(c);
            pw.flush();
        }
        catch(IOException e) {
            System.out.println("Error while writing the file...");
        }
        finally {
            pw.close();
        }
    }
}
```

Scanner

- A simple text scanner which can parse primitive types and strings using regular expressions.

Methods of **Scanner**

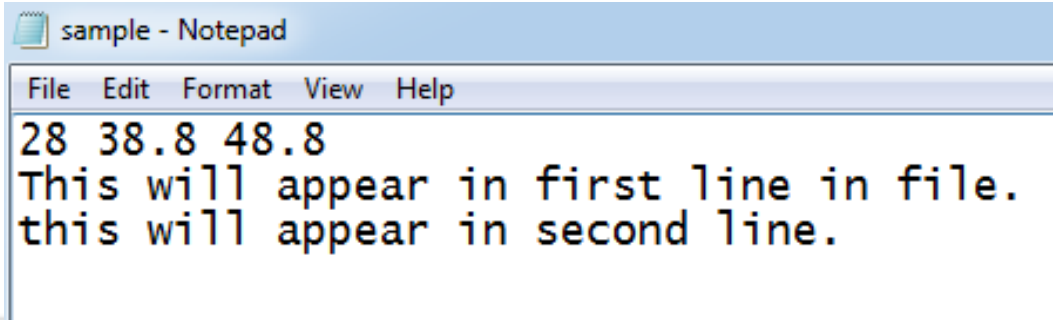
java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a Scanner that produces values scanned from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern and returns this scanner.

Example

```
import java.io.*;
import java.util.*;
class Temp {
	public static void main(String args[]) throws Exception
	{
		File loc = new File("C:/Users/hp/Desktop/sample");
		Scanner sc = null;
		try {
			sc = new Scanner(loc);
			int i = sc.nextInt();
			float f = sc.nextFloat();
			double d = sc.nextDouble();
			System.out.println(i + " " + f + " " + d);

			while(sc.hasNext()) {
				System.out.println(sc.nextLine());
			}
		}
		catch(Exception e) {
			System.out.println("Error while reading file...");
		}
		finally {
			if(sc!= null)
				sc.close();
		}
	}
}
```

Don’t pass the path of the file as String. Otherwise, it will keep on scanning the String instead of actual path of file.



Character Stream Vs Byte Stream

- In Java, characters are stored using Unicode conventions. **Character stream** automatically allows us to read/write data character by character. For example FileReader and FileWriter are character streams used to read from source and write to destination.
- **Byte streams** process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

When to use Character Stream over Byte Stream?

- In Java, characters are stored using Unicode conventions. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.

When to use Byte Stream over Character Stream?

- Byte oriented reads byte by byte. A byte stream is suitable for processing raw data like binary files.

SERIALIZATION

Serialization

- Serialization is the process of writing the state of an object to a byte stream.
- This is useful when we want to save the state of our program to a persistent storage area, such as a file.
- At a later time, we may restore these objects by using the process of de-serialization.
- Serialization is also needed to implement Remote Method Invocation (RMI).

- An object to be serialized may have references to other objects, which, in turn, have references to still more objects.

- If we attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized.

Serialization: Interfaces and Classes

- An overview of the interfaces and classes that support serialization follows:

1) Serializable

- Only an object that implements the Serializable interface can be saved and restored by the serialization facilities.
- The Serializable interface defines no members.
- It is simply used to indicate that a class may be serialized.
- If a class is serializable, all of its subclasses are also serializable.

2) ObjectOutput

- The ObjectOutput interface extends the DataOutput interface and supports object serialization.

- It defines the methods several methods. All of these methods will throw an IOException on error conditions.

- *writeObject( )* method is called to serialize an object.

3) ObjectOutputStream

- The ObjectOutputStream class extends the OutputStream class and implements the ObjectOutput interface.

- It is responsible for writing objects to a stream. Constructor is:

  ObjectOutputStream(OutputStream outStream) throws IOException

- The argument outStream is the output stream to which serialized objects will be written.

Example

```
import java.io.*;
import java.time.*;

class Employee implements Serializable
{
    String name;
    int age;
    int id;
    int salary;
    LocalDate date_of_join;

    Employee(String name, int age, int id, int salary, LocalDate doj) {
        this.name = name;
        this.age = age;
        this.salary = salary;
        this.id = id;
        this.date_of_join = doj;
    }

    public String toString() {
        String msg = "Name: " + name + " \nAge: " + age +
                     " \nSalary: " + salary + " \nID: " + id +
                     " \nDate of join: " + date_of_join;
        return msg;
    }
}
```

```java
class Test {
    public static void main(String args[]) throws IOException {
        // **********Serialization**********
        Employee e = new Employee("Rahul", 23, 11011, 25000, LocalDate.of(2017, 7, 1));
        FileOutputStream fout = null;
        ObjectOutputStream oos = null;

        try {
            System.out.println("State of object is: " + e);
            fout = new FileOutputStream("C:/Users/hp/Desktop/abc.txt");
            oos = new ObjectOutputStream(fout);
            oos.writeObject(e);
        }
        catch(IOException a)    {
            System.out.println("Error while writing object....");
        }
        finally {
            if(oos!=null)
                oos.close();
            if(fout!=null)
                fout.close();
        }
        //calling a method to perform de-Serialization
        deSerial();
    }
```

```
    static void deSerial() throws IOException
    {
        //**********de-Serialization***********
        FileInputStream fin = null;
        ObjectInputStream ois = null;
        try
        {
            fin = new FileInputStream("C:/Users/hp/Desktop/abc.txt");
            ois = new ObjectInputStream(fin);
            Employee e = (Employee)ois.readObject();
            System.out.println("State of returned object is: " + e);
        }
        catch(Exception a)
        {
            System.out.println("Error while reading object....");
        }
        finally{
            if(ois!=null)
                ois.close();
            if(fin!=null)
                fin.close();
        }
    }
}
```

Exercise

Write a program to create a class Rectangle having private attributes: length & width. Make only getter methods for them and also separate methods to return the area and perimeter.

Create a class Test having main() method in which you have to make at least 3 objects of Rectangle class (by taking the required input from user at run-time).

Perform the process of Serialization for only those objects whose area exceeds value of 100.

Also perform the process of de-serialization. Use proper exception handling in your program.

transient keyword

- **transient** is a variables modifier used in serialization. At the time of serialization, if we don't want to save value of a particular variable in a file, then we use **transient** keyword. When JVM comes across **transient** keyword, it ignores original value of the variable and save default value of that variable data type.

- **transient** keyword plays an important role to meet security constraints. There are various real-life examples where we don't want to save private data in file. Another use of **transient** keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc.

- Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use **transient** keyword with private confidential fields of a class during serialization.

Example

```
// A sample class that uses transient keyword to
// skip their serialization.
class Test implements Serializable
{
    // Making password transient for security
    private transient String password;

    // Making age transient as age is auto-
    // computable from DOB and current date.
    transient int age;

    // serialize other fields
    private String username, email;
    Date dob;

    // other code
}
```

Key points

- **transient and static :** Since **static** fields are not part of state of the object, there is no use/impact of using **transient** keyword with static variables. However there is no compilation error.

- **transient and final :** final variables are directly serialized by their values, so there is no use/impact of declaring final variable as **transient**. There is no compile-time error though.

Example

```java
// Java program to demonstrate transient keyword
// Filename Test.java
import java.io.*;
class Test implements Serializable
{
    // Normal variables
    int i = 10, j = 20;

    // Transient variables
    transient int k = 30;

    // Use of transient has no impact here
    transient static int l = 40;
    transient final int m = 50;

    public static void main(String[] args) throws Exception
    {
        Test input = new Test();

        // serialization
        FileOutputStream fos = new FileOutputStream("abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(input);

        // de-serialization
        FileInputStream fis = new FileInputStream("abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Test output = (Test)ois.readObject();
        System.out.println("i = " + output.i);
        System.out.println("j = " + output.j);
        System.out.println("k = " + output.k);
        System.out.println("l = " + output.l);
        System.out.println("m = " + output.m);
    }
}
```