7 Files and I/O Streams

Input and output (I/O) are the most commonly used operations in any program. Reading data from the keyboard and writing data on to the monitor form the basic input and output operations. It is essential that a programmer know how to read the data from files and write data to files. Java uses streams to handle data input and output. Input streams act as sources of data and output streams act as destinations of data. The java.io package defines various classes and methods for reading and writing files and handling data streams. This chapter describes various input and output streams provided in Java for reading and writing data.

7.1 An overview of I/O Streams

A flow of data is often referred to as a data stream. A stream is an ordered sequence of bytes that has a SOURCE (input stream) or a DESTINATION (output stream). In simple terms, a stream can be defined as follows:

A stream is a logical device that represents the flow of a sequence of characters.

Programs can get inputs from a data source by reading a sequence of characters from the input stream. Similarly, programs can produce outputs by writing a sequence of characters on to an output stream.

A stream can be associated with a file, an Internet resource (For example, a socket), to a pipe or a memory buffer. Streams can be chained together so that each type of stream adds its own processing to the bytes as they pass through the stream. For the purpose of letting information flow, a program opens a stream on an information source (whether it be a file, memory, a network socket or any other source) and reads the information sequentially and uses it in the program. All streams behave in the same way even though the physical device connected to them may differ. A program can send information to an external destination by opening a stream to a destination and writing the information out sequentially.

In previous chapters, we used the following two methods for taking inputs and giving outputs:

```
System.in.read()
System.out.println( );
```

These two form the basic input and output streams, which are managed by the System class; they use the standard input and standard output streams, respectively.

7.2 Java I/O

Streams are represented in Java as classes. The java.io package defines a collection of stream classes that support input and output (reading and writing). To use these classes, a program needs to import the java.io package, as shown below:

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

In general, streams are classified into two types known as *character streams* and the *byte streams*. The java.io package provides two sets of class hierarchies to handle character and byte streams for reading and writing:

1. InputStream and OutputStream classes are operated on bytes for reading and writing, respectively.
2. Classes Reader and Writer are operated on *characters* for reading and writing, respectively.

There are two other classes that are useful for handling input and output. These are File class and RandomAccessFile class.

A brief description of these four classes of java.io package are given in Table 7.1.

7.2.1 Character streams

Reader and Writer are abstract super-classes for streaming 16-bit character inputs and outputs, respectively. Methods of these classes throw the IOException exception under error conditions. All the methods in the Writer class have return type void.

Table 7.1 Description of the four classes in the package java.io.

Name of class	Description
Reader, Writer	Supports read/write 16-bit Unicode characters (specified in Java 2). Used for only text data.
InputStream, OutputStream	Define basic methods for input and output streams of data. These classes are used only for binary(byte) data.
File	Enables creating, deleting and renaming files, navigating through the file system, testing file existence and finding information about files.
RandomAccessFile	Enables the program to read/write from/to any location in the file, not just the beginning/end of the file, is the case as in the usual sequential access. The file works as a random-access disk file.

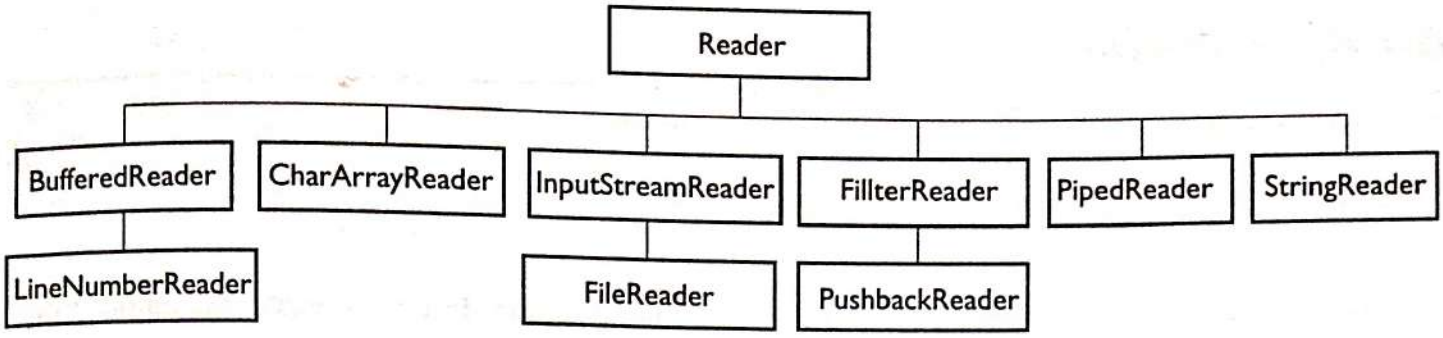


Fig. 7.1 Reader class hierarchy classes in italic are of type (i), the rest are of type (ii).

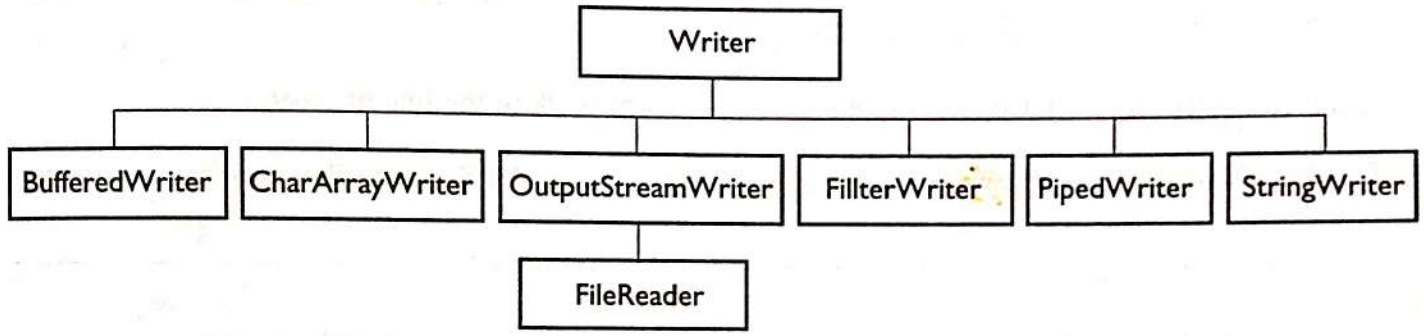


Fig. 7.2 Writer class hierarchy classes shown in static are of type (i), the rest are of type (ii).

Character streams are normally divided into two types. (i) Those that only read from or on write on to streams and (ii) those that also process the data that was read/written. Figures 7.1 and 7.2 show the class hierarchies for the Reader and Writer classes.

The descriptions of Reader and Writer classes are summarized in Table 7.2.

Table 7.2 Various Reader and Writer classes and their description.

Class name	Description
BufferedReader	Supports the buffering of the characters. It is a filter stream.
BufferedWriter	Increases the performance by buffering input/output.
CharArrayReader CharArrayWriter	Supports reading/writing of characters from/to a character array.
InputStreamReader OutputStreamWriter	Supports reading/writing of characters from/to a byte input stream.
FileReader FileWriter	Supports reading/writing of characters from/to a file. These classes uses default character encoding.

(*Table 7.2 Contd.*)

(*Table 7.2 Continued*)

Class name	Description
FilterReader FilterWriter	These two are abstract classes for characters Filter stream.
PipedReader PipedWriter	Supports the inter-thread communication. In pipe streams, the output from one method could be piped into the next one.
StringReader StringWriter	Reads/writes into the string. In StringReader the source for reading is String. Stringwriter writes the output into the StringBuffer. StringBuffer is further converted into String.
LineNumberReader	Character input stream that keeps track of the line number.
PushBackReader	Allows one-character pushback buffer for Reader. That is, after a character was read from reader, it is pushed back into the reader.

The example below illustrates how to read characters using the FileReader class.

```
FileReader fr = new FileReader("filename.txt"); //Create a FileReader class from the file
                                                filename.txt.
int i = fr.read();              //Read a character
```

Internally, to represent characters in computers, a character-encoding scheme (for example, ASCII) is generally used and every platform has a default character-encoding scheme. Java uses 16-bit Unicode character-encoding scheme to represent characters internally. The Reader classes support conversions of Unicode characters to internal character storage. Besides using default encoding, Reader and Writer classes can also specify which encoding scheme to use.

Most programs use Reader and Writer streams to read and write textual information. This is because they can handle any character in the Unicode character set. On the other hand, the byte streams are limited to ISO-Latin-1 8-bit bytes.

7.2.2 Byte streams

Byte streams are used in a program to read and write 8-bit bytes. InputStream and OutputStream are the abstract super-classes of all byte streams that have a sequential nature. InputStream and OutputStream provide the Application–Program Interface (API) and partial implementation for *input streams* (streams that read bytes) and *output* streams (streams that write bytes). These streams are typically used to read and write binary data such as those related to images and sounds. Methods of these two classes throw the IOException. All methods of the OutputStream will have the return type void.

The hierarchies of InputStream class and OutputStream class are shown in Figures 7.3 and 7.4. All the sub-classes of InputStream and OutputStream work only on bytes. Note that both

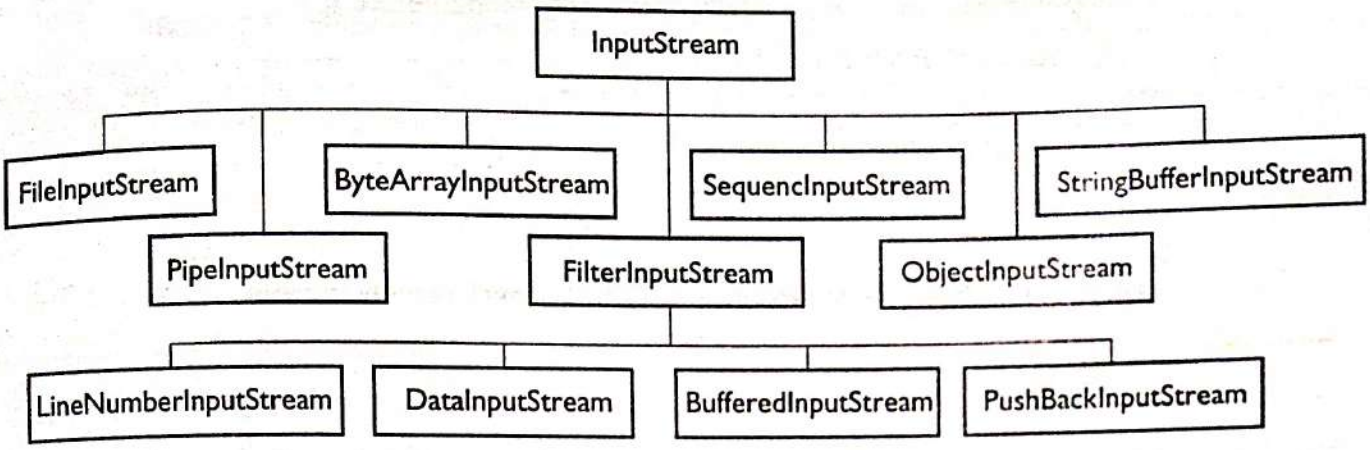


Fig. 7.3 InputStream class hierarchy.

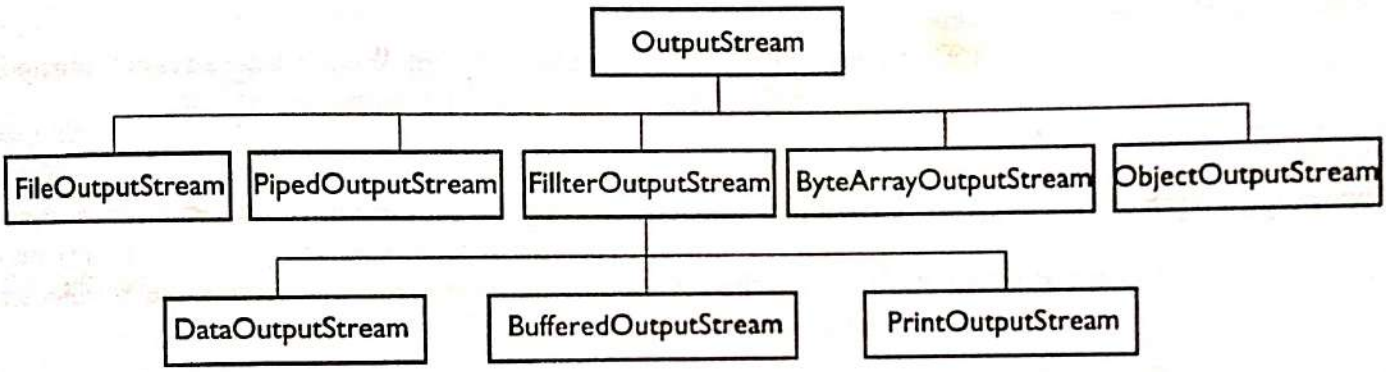


Fig 7.4 OutputStream class hierarchy.

InputStream and OutputStream are inherited from the Object class. Since InputStream and OutputStream are abstract classes, they cannot be used directly.

A brief description of classes in the InputStream and OutputStream hierarchies are given in Table 7.3.

Two other classes that are available are ObjectInputStream and ObjectOutputStream, which are used for object serialization. These are the sub-classes of InputStream and OuputStream which internally implement the ObjectInput and ObjectOutput interfaces, respectively. These classes are covered in section 7.7. We cover the most useful stream classes in the rest of this chapter.

7.2.3 Working with the I/O super-classes

The classes Reader and InputStream define similar APIs but for different data types. Reader contains the methods described in Table 7.4 for reading characters and arrays of characters.

Similarly, InputStream defines the same methods but for reading bytes and arrays of bytes. These are listed in Table 7.5.

Both Reader and InputStream provide methods for marking a location in the stream, skipping input and resetting the current position. The following code illustrates reading a character.

```
FileInputStream inp = new FileInputStream("filename.txt");
while ((input = inp.read()) != null)
{
    System.out.println(input);
}
```

Table 7.3 Classes in the InputStream and OutputStream hierarchies.

Name of class	Description
FileInputStream and FileOutputStream	Read/write data from/to a file on the native file system. These work only on bytes. FileOutputStream allows creating a file if the file does not exist.
FilterInputStream and FilterOutputStream	Abstract streams that are used to add new behaviours to existing stream classes. Used to read from or write to another stream. When a filter stream is created, it must be specified as to which stream it attaches to.
PipedInputSteam and PipedOutputStream	Used for inter-thread communications. These classes implement the input and output components of a pipe. Pipes are used to channel the output from one program (or thread) into the input of another. A PipedInputStream must be connected to another PipedInputStream.
ByteArrayInputStream and ByteArrayOutputStream	Read data from or write data to a byte array in memory.
SequenceInputStream	Concatenate multiple input streams into one input stream
StringBufferInputStream	Allow programs to read from a StringBuffer as if it were an input stream.
DataInputStream and DataOutputStream	Allows programs to read and write primitive Java data types such as int, long, boolean and char in a machine-independent format.
BufferedInputStream and BufferedOutputStream	Allows buffering of the data during reading or writing. These classes are used to buffer the data to speed up the reading and writing process. Buffering reduces the number of accesses required on the original data source and thus increasing the speed of the process.
LineNumberInputStream	Input stream that keeps track of line numbers while reading.
PushbackInputStream	Allows one-byte pushback buffer for an input stream. That is, after a byte was read from an input stream, it is pushed back on to the input stream.
PrintStream	An output stream used to display the text. It allows the program to print the output in a fashion convenient to an output stream.

Table 7.4 Methods contained by the Reader class, for reading character arrays.

Name of method	Description
int read( )	Returns the integer specifying the next available character in the input stream. Otherwise, it returns –1, that is, the end of the file.
int read(char cbuf[ ])	Allows reading up to the buffer and returns the number of characters read. Otherwise, it returns –1.
int read(char cbuf[ ], int offset, int length)	Attempts to read number of characters specified by length from the offset. Otherwise, it returns –1.

Table 7.5 Methods contained by the InputStream class, for reaching arrays of bytes.

Name of Method	Description
int read( )	Returns integer specifying the next available byte in the input stream. Otherwise, it returns –1, that is the end of the file.
int read(char cbuf[ ])	Allows reading up to the buffer and returns the number of bytes read. Otherwise, it returns –1.
int read(char cbuf[ ], int offset, int length)	Attempts to read the number of bytes specified by length from the offset. Otherwise, it returns –1.

Similarly, Writer and OutputStream are parallel concepts. Like Reader and InputStream, Writer (or OutputStream) defines the following methods for writing characters (or bytes) and arrays of characters (or arrays of bytes):

```
void write( int c )
void write( char cbuf[ ] )
void write( char cbuf[ ], int offset, int length )
```

These methods are used to write into the invoking stream. The first method writes a single character (or byte, in the case of OutputStream) to the invoking output stream. The second method writes the complete array into the invoking output stream. The final method writes the sub-range of the length of characters (or bytes, in the case of output stream) starting from the offset value of the buffer to the invoking stream.

All of these stream classes, namely, Reader, Writer, InputStream and OutputStream are automatically opened when they are created. A stream can be closed either implicitly or explicitly. When the stream object is no longer referenced, the garbage collector can implicitly close it. Alternatively, the close() method can be used to close the stream explicitly.

In addition to read(), write() and close() methods, Table 7.6 lists some other methods that belong to InputStream and OutputStream classes.

Table 7.6 Other methods in InputStream and OutputStream classes.

Name of method	Description
InputStream	
long skip(long n)	Skips *n* bytes in the input stream. It returns the number of bytes skipped.
int available()	Returns the number of bytes still available in the input stream.
Synchronized void reset()	Returns the file handler to the marked position that was previously set in the stream.
Synchronized void mark(int n)	Marks a position in the stream that will be valid till the next *n* bytes are read.
boolean markSupported()	Returns true if the stream supports mark or reset, otherwise returns false.
OutputStream	
Void flush()	Flushes any buffered output to be written.

7.3 File Streams

A file can be created using File class.

```
File f;
f = new File (string filename);
```

The File class is defined in the package java.io The File class is used to store the path and name of a directory or file. But this class is not useful in retrieving or storing data. The File object can be used to create, rename or delete file or directory it represents. A directory is treated as a file that contains a list of files. Each file (or directory) in Java is an object of the File class. A File object is used to manipulate the information associated with a disk file, such as last modification date and time, directory path and also to navigate through the hierarchies of sub-directories.

The File class has three constructors that are used to create a file object. They are the following:

- File(String pathname); // pathname could be file or a directory name
- File(String dirPathname, String filename); // specify directory where the file is created // and file name
- File(File directory, String filename); // file object as the directory path

These constructors return the reference to the file object.

The File class provides several methods that deal with the files, file system, properties of the file and tests on the files. Some of the important methods that serve these purposes are listed in Table 7.7.

Table 7.7 Some methods in the File class and their description.

Name of method	Description
string getName()	Returns the name of the file excluding the directory name.
string getPath()	Returns the path of the file.
string getParent()	Returns the parent directory of the file.
boolean exists()	Returns true if the file exists and returns false if the file does not exist.
boolean canRead()	Returns true if the file is readable, otherwise returns false.
boolean canWrite()	Returns true if the file is writable, otherwise returns false.
boolean isFile()	Returns true if the object is a file, otherwise returns false.
boolean isDirectory()	Returns true if the object is a directory, otherwise returns false.
boolean isAbsolute()	Returns true if the file path is the absolute path, otherwise it returns false, indicating that it is a relative path.
long length()	Returns the length of the file (in bytes).
long lastModified()	Returns the date on which the file was last modified
String[] list()	Returns the list of all files and directories that exist in the current directory

There are other methods also which are useful for handling file operations such as renaming a the file and deleting a file. The syntax of these two methods is the following:

```
boolean  renameTo(File new file name);
boolean delete();
```

The following methods are used to create a directory:

```
boolean mkdir(File  newdirectoryname)
boolean mkdirs(File newdirectoryname)
```

The first method creates a directory and returns true if it is successful. If the directory cannot be created, it returns false. In addition to creating a directory, the second method creates all its parent directories.

The path of the directly to be created will be passed as a parameter to the mkdirs method. If the program fails to execute the mkdirs() method fully, it may be successful in creating partial directories list from the user specified path. The method mkdirs() returns true if the directory was created with all the necessary parent directories, otherwise it returns false.

Directory is the name of an object in the File class with an extra property referred to as 'list' which is examined by the list() method. This method returns the string array with all files that exist in the directory. Program 7.1 illustrates the list() method.

Program 7.1 Using the methods list() and isDirectory().

```
import java.io.*;

class ListOfFiles
{
    public static void main(String a[ ])
    {
        String name = arg[0];
        File file1= new File(name);
        if ( file1.isDirectory() )
        {
            string  arr[ ] = file1.list();
            for (int i  = 0; i < arr.length; i++)
            {
                if(arr[i].isDirectory())
                    System.out.println(arr[i]+"is directory");
                else
                    System.out.println(arr[i]+"is file");
            }
        }
    }
}
```

Program 7.1 illustrates the isDirectory() method, which returns the value true if the invoking file object is a directory and returns the value false if it is a file. This program is run with the command line arguments in which arg[0] specifies the name of the directory. The output of the program consists of a list of all the files and directories in the present directory that is, the directory given as the input.

7.4 FileInputStream and FileOutputStream

The File class explains only the file system. Java provides two special types of stream called the FileInputStream and FileOutputStream to read data from and write data into the file. These classes operate on the files in the native file system. FileInputStream and FileOutputStream are sub-classes of InputStream and OutputStream, respectively. Usually, we use FileInputStream (for byte streams) or FileReader (for character streams) for reading from a file and FileOutputStream (for byte streams) or FileWriter (for character streams) for writing into a file.

We can create a file stream by giving the file name as a parameter in the form of a string, File object or FileDescriptor object. FileWriter and FileOutputStream will create a new file by that name if the file does not already exist.

The constructors of these two streams are:

- FileInputStream(String filename);
- FileInputStream(File fileobject);
- FileInputStream(FileDescriptor fdesobject);
- FileOutputStream(String filename);
- FileOutputStream(File fileobject);
- FileOutputStream(FileDescriptor fdesobject);

FileDescriptor is an object that holds the information about a file.
A FileInputStream object can be created in the following manner:

```
FileInputStream fin = new FileInputStream(string filename);
```

Alternatively, it can be created with the following set of statements:

```
File f = new File(string filename);
FileInputStream fin = new FileInputStream(f);
```

A FileOutputStream object can be created as follows:

```
FileOutputStream fout = new FileOutputStream(string filename);
```

Note that in the case of FileOutputStream if it is opened on a file that does not already exist then a new file by that name is created.

Program 7.2 implementing the class CopyFile uses FileInputStream and FileOutputStream to copy the contents of inputFile to outputFile: inputFile and outputFile are the command line arguments arg[0] and arg[1].

Program 7.2 Using CopyFile, FileInputStream and FileOutputStream.

```
import java.io.*;
public class CopyFile
{
    public static void main(String[ ] args) throws IOException
    {
        if(args.length<2)
        System.out.println("\n No sufficient parameters");
        else
        {
            File inputFile = new File(args[0]);
            File outputFile = new File(args[1]);
            FileInputStream in = new FileInputStream(inputFile);
            FileOutputStream out = new FileOutputStream(outputFile);
            int c;
            while ((c = in.read()) != -1)
            out.write(c);
```

```
            in.close();
            out.close();
        }
    }
}
```

In Program 7.2, the file inputFile is opened using FileInputStream and another file outputFile is opened using FileOutputStream. If inputFile does not exist, the program gives an error because FileInputStream cannot work if the file does not exist. On the other hand, if outputFile does not exist, then a file by that name will be created. The program continues to read characters from FileInputStream as long as there are inputs in the input file and writes these characters on to FileOutputStream. When all the inputs have been read, the program closes both FileInputStream and FileOutputStream. Program 7.2 can also be written using FileReader and FileWriter with the small changes shown below:

```
File inputFile = new File(args[0]);
FileReader in = new FileReader(inputFile);
```

This code creates a File object that represents the named file on the native file system. File is a utility class provided by java.io. Program 7.2, the CopyFile program uses this file object only to construct a FileReader on a file; however, the program could also use the input file to get information about the file, such as its full path name.

If we run Program 7.2, an exact copy of inputFile (args[0]) will be found in a file named outputFile (args[1]) in the same directory. It should be remembered that FileReader and FileWriter read and write 16-bit characters; however, most native file systems are based on 8-bit bytes. These streams encode the characters as they operate according to the default character-encoding scheme. The default character-encoding scheme can be found by using System.getProperty("file.encoding"). To specify an encoding scheme other than the default, we should construct an OutputStreamWriter on a FileOutputStream and specify the encoding scheme.

It is important to note that the FileOutputStream object does not support the means to append a file. If the file exists already, FileOutputStream only overwrites and cannot add content to an end of the file.

7.5 Filter Streams

The java.io package provides a set of abstract classes that define and partially implement *Filter streams*. A Filter stream filters data as it is being read from or written to the stream. The two filter streams for reading and writing data are FilterInputStream and FilterOutputStream, respectively.

A filter stream is constructed on another stream. That is, every filtered stream must be attached to another stream. This can be achieved by passing an instance of InputStream or OutputStream to a constructor. This implies that filter streams are chained. Multiple filters can be added by chaining them to a stream of bytes.

The read method in a readable filter stream reads input from the underlying stream, filters it and passes on the filtered data to the caller. The write method in a writable filter stream filters the data and then writes it to the underlying stream. The filtering done by the streams depends on the stream. Some streams buffer the data, some count data as they go by and others convert data from the original form to a different one.

Most filter streams provided by the java.io package are sub-classes of FilterInputStream and FilterOutputStream, and are listed below:

- DataInputStream and DataOutputStream
- BufferedInputStream and BufferedOutputStream
- LineNumberInputStream
- PushbackInputStream
- PrintStream

The java.io package contains only one sub-class of FilterReader known as PushbackReader.

To create filter input or output stream, we should attach the filter stream to another input or output stream. For example, we can attach a filter stream to the standard input stream in the following manner:

```
BufferedReader d = new BufferedReader(new InputStreamReader(System.in));
String input;
while ((input = d.readLine()) != null)
{
    .....
}
```

The above code is useful for reading the data from the console. Here, InputStreamReader is attached to BufferedReader which acts as the filter stream.

7.5.1 DataInputStream and DataOutputStream

DataInputStream (DataOutputStream) is a filtered input (output) stream and hence must be attached to some other input (output) stream. DataInputStream handles reading in data as well as lines of text.

Program 7.3 stores the data which are read from the console in tabular format using DataOutputStream. After that, the program reads the data using the DataInputStream.

Program 7.3 Using DataOutputStream and DataInputStream.

```
import java.io.*;
class SampleStream
{
    public static void main(String a[ ]) throws IOException
```

```
{
    DataOutputStream out = new DataOutputStream(new
                                FileOutputStream("file.txt"));
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    char sep = System.getProperty("file.separator").charAt(0);
    for (int i = 0; i < 5; i ++)
    {
        System.out.print("Read the item name:");
        String item=br.readLine();
        out.writeChars(item+separator);
        out.writeChar('\t');
        System.out.print("Read the quantity:");
        int qun=Integer.parseInt(br.readLine());
        out.writeInt(qun);
        out.writeChar('\n');
    }
    out.close();
    DataInputStream in = new DataInputStream(new
    FileInputStream("file.txt"));
    try
    {
        char chr;
        while (in.available()>0)
        {
            StringBuffer item1 = new StringBuffer(20);
            while ((chr = in.readChar())!= sep)
            {
                item1.append(chr);
            }
            System.out.print(item1);
            System.out.print(in.readChar());
            System.out.print(in.readInt());
            System.out.print(in.readChar());
        }
    }
    catch (EOFException e)
    {
    }
```

```
            in.close();
        }
    }
```

Running Program 7.3 gives the following output:

```
Read the item name: Mouse
Read the quantity: 4
Read the item name: Keyboard
Read the quantity: 3
Read the item name: Monitor
Read the quantity: 2
Read the item name: Modem
Read the quantity: 4
Read the item name: Hub
Read the quantity: 4

Mouse       4
Keyboard    3
Monitor     2
Modem       4
Hub         4
```

The above example stores the items required and the quantity of the item. Although the data are read or written in binary form, they appear on the console as characters.

DataOutputStream is attached to a FileOutputStream that is set up to write to a file named file.txt. DataOutputStream writes the data in the file.txt file by using specialized writeXXX() methods. Here, XXX represents a Java data type. Next, SampleStream opens a DataInputStream on the file just written to read the data from it by using DataInputStream's specialized readXXX() methods.

In general, we use null or –1 to indicate the end of the file. Some methods such as readString() and readChar() in the DataInputStream do not identify null or –1 as an end of file marker. Thus, a programmer must be careful while writing loops using Data Input stream for reading characters from a file.

The available() method of DataInputStream, which returns the number of bytes remaining, can be used to check whether there are any data remaining to read.

7.5.2 PushbackInputStream

Pushback is used on input streams to allow a byte to be read from and returned to the stream. Like all filter streams, PushbackInputStream is also attached to another stream. It is typically used for

some specific implementation, for example, to read the first byte of the data from a stream for the purpose of testing the data.

For example, it is mostly required in the case of designing parsers, while reading data, to have a glance on the next set of input data. Data must be read, and if not found appropriate, it should be placed back into the input stream, which may be read again and processed normally.

A PushbackInputStream object can be created as follows:

```
PushbackInputStream inpb = new PushbackInputStream(new InputStream);
```

In order to satisfy the requirement of pushing back a byte (or bytes) of an array into the input stream, this stream has the special method called the unread() method.

```
void unread( int ch );
void unread( bute buf[ ] );
void unread( byte buf[ ], int offset, int numchars );
```

The first form pushes back the lowest-order byte of ch, which is returned as the next byte when another byte is read. The second and the third forms of the method return the byte in a buffer with the difference that the third method pushes back the number of characters (numchars) starting from the offset. PushbackReader also does the same thing on the character stream.

7.6 RandomAccessFile

Sequential files can read/write only at the beginning/end of the file. On the other hand, random access files allow us to read from or write to any location in the file. The class RandomAccessFile offers methods that allow specified mode accesses such as 'read only' and 'read – write' to files.

Since RandomAccessFile is not inherited from InputStream or Output Stream it cannot be chained unlike filtered stream classes; however, RandomAccessFile implements both DataInput and DataOutput which deal with Java's primitive data types. It must be noted that DataInputStream implements DataInput whereas DataOutputStream implements DataOutput.

The RandomAccessFile class is a very useful class for file handling. The constructors of these classes are the following:

```
RandomAccessFile(String filename, String mode);
RandomAccessFile(File file, String mode);
```

Thus, RandomAccessFile objects can be created either from a string containing the file name or from a File object. The mode represents the type of access to the file, for example, 'r' for read and 'rw' for read and write. There are two other modes, namely, 'rws' and 'rwd', 'rws' opens the file as read and write and stores the updations made on the file data or metadata in the synchronized way, whereas, 'rwd' deals with the file data but not the metadata. Here, metadata refers to the data about the files. RandomAccessFile throws an IOException if an I/O error has occurred. Program 7.4 illustrates the use of RandomAccessFile.

Program 7.4 Using RandomAccessFile.

```
import java.io.*;

class RandomAccessExample
{
    public void method_B()
    {
        try
        {
            RandomAccessFile rafile = new RandomAccessFile("file.txt","rw");
            System.out.print("Content of the file is :");
            for ( int i = 0; I < rafile.length() –1; i++)
            {
                if ( (i%2) == 0)
                {
                    rafile.seek(i);
                    char myc = rafile.readChar();
                    System.out.print(myc);
                }
            }
            rafile.seek(rafile.length());
            rafile.writeChars(" Append");
            System.out.println();
            System.out.print("Content of the file is after append:");
            for (int i =0; i < rafile.length() –1; i++ )
            {
                if ( (i%2) == 0)
                {
                    rafile.seek(i);
                    char myc = rafile.readChar();
                    System.out.print(myc);
                }
            }
        }
        catch(IOException ioe)
        {
            System.out.println("error:" + ioe.getMessage());
        }
    }
```

```
    void method_A()
    {
        try
        {
            String s = new String("RandomAccess Example");
            FileOutputStream fos = new FileOutputStream("file.txt");
            DataOutputStream dos = new DataOutputStream(fos);
            dos.writeChars(s);
        }
        catch(IOException ioe)
        {
            System.out.println("error:"+ioe.getMessage());
        }
    }
    public static void main(String args[])
    {
        RandomAccessExample rae = new RandomAccessExample();
        rae.method_A();
        rae.method_B();
    }
}
```

The output of Program 7.4 is the following:

```
Content of the file is: RandomAccess Example
Content of the file is: RandomAccess Example Append.
```

In Program 7.4, the method seek(long number) points the file pointer to the location specified by number.

7.7 Serialization

Serialization is the technique most commonly used for network applications and persistent storage of objects, in order to maintain the reference of a remote object.

Storing the state of the object into a file is often referred to as *serialization*. whereas, reading the object state from a stored file is referred to as *deserialization*.

The streams ObjectOutputStream and the ObjectInputStream are used for object serialization and deserialization, respectively. These classes allow us to read/write objects from to streams converting from internal to external 8-bit representation. These two classes throw the exceptions IOException and StreamCorruptedException.

The two methods that are useful for object serialization are readObject() on ObjectInputStream and writeObject(Object *obj*) on the ObjectOutputStream. To make any object serializable, the object of the class must implement the serializable interface of the java.io package. That is, we have to specify implements Serializable in the definition of the class, the objects of which we would like to read/write from to files.

Any variable in a class having the *transient* modifier will not be saved in the file (that is, that variable is not serializable). Similarly static variables are not serializable. There is an interface called Externalizable, which is used to control the serialization process of the object.

Program 7.5 implementing the SerializationExample class illustrates the use of the object serialization, which stores the object state in a file called object.txt and deserializes it.

Program 7.5 Using object serialization and deserialization.

```
import java.io.*;
class ObjectSerialization implements Serializable
{
    String Employeename;
    String designation;
    ObjectSerialization (String number,String designation)
    {
        this.Employeename=number;
        this.designation =designation;
    }
    public String toString()
    {
        System.out.println("Employee name is:"+Employeename);
        System.out.println("Designation is:"+designation);
    }
}
public class SerializationExample
{
    public static void main(String args[ ])
    {
        try
        {
            ObjectSerialization obj = new ObjectSerialization ("Kumar",
                                                    "ProjectLeader");
            FileOutputStream fos=new FileOutputStream("object.txt");
            ObjectOutputStream oos=new ObjectOutputStream(fos);
```

```
            oos.writeObject(obj);
            oos.flush();
            oos.close();
            FileInputStream fis=new FileInputStream("object.txt");
            ObjectInputStream ois=new ObjectInputStream(fis);
            ObjectSerialization obj2=new (ObjectSerialization) ois.readObject()
            System.out.println("value of the obj2: "+obj2);
        }
        catch(NotSerializableException e1)
        {
            System.out.println("ioexception has occurred:"+e1.getMessage());
        }
        catch(StreamCorruptedException e2)
        {
            System.out.println("ioexception has occurred:"+e2.getMessage());
        }
        catch(IOException e3)
        {
            System.out.println("ioexception has occurred:"+e3.getMessage());
        }
    }
}
```

The output of Program 7.5 is the following:

```
Value of the obj2:
Employee name is: Kumar
Designation: ProjectLeader
```

FileInputStream and FileOutputStream are used to read/write the object from/to a file, respectively. In Program 7.5, when we have problems with stream classes, the exceptions caught by the try block may be IOException or StreamCorruptedException. The exception NotSerializableException will arise if the class that is going to be serializable does not implement the serialization interface.

(Sample programs involving concepts introduced in this chapter can be found at http://www.universitiespress.com/downloads/books/81-7371-572-6.pdf, section 7.8 Sample Programs)

Objective type questions

1. _____ is a logical entity that either produces or consumes information.
2. The _____ and _____ are abstract classes used to handle character streams.
3. The _____ class tops the hierarchy of input byte-stream classes.
4. When working with files the _____ package is imported.
5. The read method in FileInputStream class returns –1 when _____ occurs.
6. FileInputStream class throws _____ exception apart from IOException.
7. The static stream objects 'out'and 'err'of System class are actually objects of type _____.
8. All the methods in Writer class have the _____ return type.
9. The _____ class allows reading of binary representations of Java primitives from an input byte stream.
10. Which encoding scheme does Java support to represent characters internally?
    a. 16-bit Unicode b. 8-bit Unicode c. 8-bit ASCII d. 16-bit ASCII
11. What happens when the following code is executed?

```
import java.io.*;
public class Question12
{
    public static void main(String args[])
    {
        InputStream is=new InputStream("test.txt");
        int i=is.read();
        while(i!=–1)
        {
            System.out.println(i);
            i=is.read();
        }
        is.close();
    }
}
```

    a. Prints the contents of file test.txt to standard output.
    b. The code will not compile because the 'unreported exception java.io.IOException; must be caught or declared to be thrown'.
    c. The code will not compile because the 'unreported exception java.io.FileNotFoundExcept ion; must be caught or declared to be thrown.'
    d. The code will not compile because abstract class InputStream cannot be instantiated.

8

Applets

An applet is a software component that enables client-side programming and facilitates text, graphics, audio, imaging, animation and networking, besides live updating and securing two-way interaction in web pages. This chapter introduces applets, applet methods and applet life-cycle.

8.1 Introduction

One of the main features of Java is the applet. Applets are dynamic and interactive programs. Applets are usually small in size and facilitate event-driven applications that can be transported over the web.

An applet is a Java code that must be executed within another program. It mostly executes in a Java-enabled web browser. An applet can be embedded in an HTML document using the tag <APPLET> and can be invoked by any web browser. Parameters can be passed to an applet using the tag <PARAM> in the HTML document. Applets can also be executed using the appletviewer utility provided in the J2SDK kit.

The Applet class is defined in the package java.applet. All the basic capabilities of applets are provided in this class. To write Java applets, the class Applet present in the java.applet package must be extended. The Applet class has to be imported as shown below:

```
import  java.applet.Applet
```

A Java applet must be a public class. Thus, the Applet class can be accessed when the applet is run in a web browser or in the Appletviewer. The applet code will be compiled even if the class is not declared as public, but the applet will not run.

An applet code looks as shown below:

```
import java.applet.*;
public class SampleApplet extends Applet
{
    // applet code
}
```

Look at the following simple applet code that displays a string:

```
import java.applet.Applet;
import java.awt.Graphics;

public class SimpleApplet extends Applet
```

```
{
    public void paint (Graphics g)
    {
        g.drawString ("Simple Applet Code"), 30, 45);
    }
}
```

Like any other Java program, the applet's class name must match with the name of the file in which it is written.

Applets do not contain the main() that is required to run a Java application and thus they cannot run on its own; however, usually applets run in a container that provides the missing code. The main features of applet include the following:

- It provides a GUI
- facilitates graphics, animation and multimedia
- provides a facility for frames
- enables a event handling
- avoids a risk and provides a secure two-way interaction between web pages.

Note that it is possible to execute applets using the Java interpreter (using the main() method) if the class is extended with Frame.

An applet can reside anywhere on the web and can be downloaded on any computer. Applets have limited access to the resources of the client into which they are installed from an Internet server. For this reason, applets avoid the risk of viruses or breaking the data integrity problems at the client. Applets provide security by reverifying the generated byte code and no memory space is allocated till the reverification process completes. Generally, a virus cannot be hidden in uninitialized memory. If the reverification process is not complete, the applet will be rejected. Further, applets cannot do the following:

- Read or write client file system (disks) unless specified explicitly; in such cases, the applet is allowed only in certain directories
- Access the source code of the applets; the applet's source code can be accessed only from the original server
- Execute local programs/libraries/ DLLs on the client
- Opening network connections other than to a HTTP server: applets only talk to the server that they live on
- Finding information about the user such as user name, directories and applications installed

Applets are very useful for the presentation and demonstration of a concept in an effective manner. For example, an applet designed for explaining the bubble sort algorithm shows visually how the bubble is moved during each pass of the algorithm. Usually, applets interact with the AWT controls and it is necessary to import the awt package (java.awt.*) and also the event package

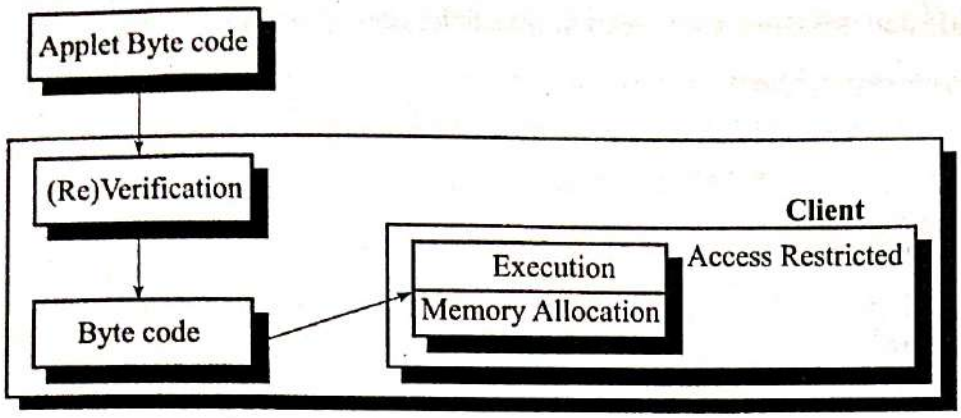


Fig. 8.1 Byte code of an applet running at a client.

(java.awt.event.*) for event handling. Input/Output streams are not commonly used for applets for the purpose of I/O. Figure 8.1 illustrates the byte code of an applet running at a client.

8.2 Java Applications Versus Java Applets

Java programs consist of applications and applets. Java applications are stand-alone programs and typically executed using the Java interpreter. On the other hand, Java applets are executed in a web browser or in a special window known as the Appletviewer; however, the compilation stages of both are the same.

The structure of applets differs from that of application programs; for example applets do not have the main() method. Applets do not interact with the console operations (that is, I/O operations) like application programs do. Applets allow the user to interact through the AWT .

Applets differ from applications in the way they are executed. An application is started when its main() method is called. In the case of applets, an HTML file has to be created, which tells the browser what to load and how to run the applet. Java, however, allows a programmer to create software code that can function both as a Java application and as a Java applet using a single class file. Table 8.1 illustrates the similarities and differences between Java applications and applets.

8.3 Applet Life-cycle

Applets need not be constructed externally. The run–time environments associated with the applets context, either the browser or the applet viewer, automatically construct them. Every applet must extend the Applet class. Unlike application programs, Java applets have a special programming structure.

An applet has a life-cycle, which describes how it starts, how it operates and how it ends. Figure 8.2 shows the life-cycle of an applet. The life-cycle consists of four methods: init(), start(), stop() and destroy(). These methods are used as follows:

- **init()** It is called only once when the applet is first loaded and created by the browser.
- **start()** It runs whenever the applet becomes visible.

Table 8.1 Similarities and differences between applications and applets.

Java applications	Java applets
These run on stand-alone systems.	These run in web pages.
These run from the command line of a computer	These are executed using a web browser.
Parameters to the application are given at the command prompt (for example, args[0], args[1] and so on)	Parameters to the applet are given in the HTML file.
In an application, the program starts at the main() method. The main() method runs throughout the application	In an applet, there is no main() method that is executed continuously throughout the life of an applet.
These have security restrictions.	These have security restrictions.
They support GUI features.	They support GUI features too. Java–compatible browsers provide capabilities for graphics, imaging, event handling and networking.
These are compiled using the javac command	These are compiled using the javac command also.
These are run by specifying at the command prompt as follows: `java classFileName`	These are run by specifying the URL in a web browser, which opens the HTML file. Or run using the appletviewer by specifying at the command prompt: `appletviewer htmlFileName`

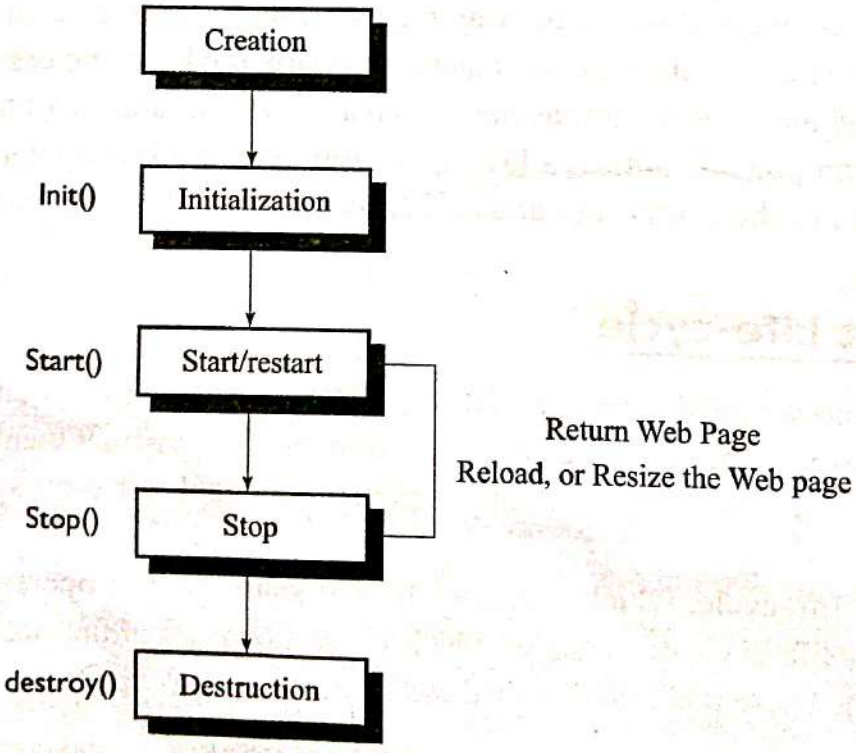


Fig. 8.2 Applet life-cycle.

- stop() It is called when the applet becomes invisible.
- destroy() It is called when the applet is shutdown.

A better understanding of these methods and how they reutilized in the applet can be got from the following description of the applets life-cycle.

In addition, there is a method paint() which is used to draw the applet for display, Note that the paint() method is not actually part of applet life-cycle as defined in Figure 8.2, but without display an applet is not always useful. Hence it is understood to be a part of the life-cycle.

When an applet is loaded (or reloaded), the browser creates an instance of the applet into a browser and then initializes it before starting the applet. When a user leaves (or minimizes) a web page, the browser stops the applet and when the user returns to the page the browser starts the applet again.

The default implementations of the init(), start(), stop(), destroy() and paint() methods do not have any code and, thus, do nothing at all. All the life-cycle methods can be overridden to add specific features of applet program. The following is a brief description of the life-cycle methods:

8.3.1 The init() method

The syntax for using init() is the following:

```
public void init { .... }
```

The init() method of the Applet class provides the capability to initialize the applet variables. That is, this method can be overridden to perform basic initialization of an applet. The method init() is called when the applet is created and loaded for the first time into a browser (or reloaded). This method is called only once in the applet's life-cycle.

The browser calls the init() method after the constructor of the applet is called. Usually, most of initialization is done in the init() method instead of the constructor. This is because, the constructor may be called before the hosting program is ready to provide all of the services needed for initialization.

8.3.2 The start() method

The syntax for using start() is the following:

```
public void start() { .... }
```

The start() method is called to start (or restart) the execution of the applet The browser can call this method after the init() method is completed.

The start() method runs whenever the applet becomes visible That is, this method is called each time the applet is going to be displayed on the screen. For example, while navigating through the web, whenever a user returns to a web page in which the applet is written, the applet is restarted. Thus, unlike the init() method, an applet can start many times throughout its life-cycle.

The start() method can also be called if the applet is stopped and to start animation and play sounds. A single start() method can start up one or several threads.

8.3.3 The stop() method

The syntax for the stop() method is the following:

```
public void stop() { .... }
```

The stop() method stops the applet and makes it invisible. This method can also be used to stop specific features of an applet, such as animation. For example, the stop() method runs when the user leaves the current web page. An applet can be invisible typically when a user moves to another web page or when the browser is minimized. The stop() method keep the applet running in the background. The stop() method also stops all the threads running on the applet.

8.3.4 The destroy() method

The syntax for the destroy() method is the following:

```
public void destroy(){ .... }
```

The destroy() method is called to terminate an applet. This method runs either before the applet exits or before the browser exits. It is useful for clean-up actions, such as releasing memory after the applet is removed, killing off threads and closing network/database connections. Thus, this method releases all the resources that were initialized during an applet's initialization.

When the destroy() method is called, the applet is removed from the memory. A call to the destroy() method will call the stop() method to execute. This method can be overridden if any specific resources have to be released.

8.3.5 The paint() method

The syntax for the paint() method is the following:

```
public void paint(Graphics g) { .... }
```

The paint() method is used for applet display on the screen. The display includes text, images, graphics and background. The paint() method is called whenever a window is required to paint or repaint the applet. It can also be called when a browser window is maximized or when the applet is shown that was covered by another window. Thus, this method can be called several times during an applet's life-cycle. In general, every applet program will have a paint() method, which is overridden to display the applet. To override paint(), we have to import the java.awt.* libraries, which include the Graphics class that enables display of information and graphics in an applet's display area.

Unlike the other four methods of an applet, paint() has one argument of type Graphics. Hence, to use paint(), the graphics class must be imported. Whenever paint() is called, an instance of

Table 8.2 Applet methods.

Name of method	Features
init()	The first method to be called. Runs once at the time of initialization before the applet starts. That is, it is called when the applet is first loaded and created by the browser. This method is employed to initialize variables. It is also used to load images and fonts.
start()	Runs whenever the applet becomes visible. That is, it is called when an applet starts or restarts after being stopped. Occurs after init().
stop()	Runs whenever the applet becomes invisible. That is, it is called when the applet leaves the web page.
destroy()	Runs only once when the browser exits. The applet will be removed from the memory. Reclaims resources that were allotted during initialization and, thus, it is used for deallocating, closing and cleaning up resources.
Paint()	Runs whenever the applet needs to be drawn or displayed.

the graphics class is created by the browser and handed over to the applet. Further, paint() is not defined in the Applet class. Applet inherits paint() from the Component class, a super-class in Applet's long chain of inheritance, which goes from Applet to Panel to Container and finally to Component.

The repaint() method is called if the applet wants to be repainted again. Note that repaint() internally calls paint().

A brief summary of sections 8.3.1 to 8.3.5 is presented in Table 8.2.

8.4 Working with Applets

Applets can be executed in two ways : from Browser or from Appletviewer. The JDK provides the Appletviewer utility.

Browsers allow many applets on a single page, whereas applet viewers show the applets in separate windows.

Program 8.1, named SampleApplet, demonstrates the life-cycle methods of applets:

Program 8.1 A demonstration of life-cycle methods of applets.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

```
/*<APPLET code = "SampleApplet" width=300 height=150> </APPLET> */
public class SampleApplet extends Applet implements ActionListener
{
    Button b=new Button("Click Me");
    boolean flag=false;
    public void init()
    {
        System.out.println("init() is called");
        add(b);
        b.addActionListener(this);
    }
    public void start()
    {
        System.out.println("start() is called");
    }
    public void destroy()
    {
        System.out.println("destroy() is called");
    }
    public void stop()
    {
        System.out.println("stop() is called");
    }
    public void actionPerformed(ActionEvent ae)
    {
        flag=true;
        repaint();
    }
    public void paint(Graphics g)
    {
        System.out.println("paint() is called");
        If(flag)
        g.drawString("This is a simple Applet",50,50);
    }
}
```

For executing the program using the AppletViewer or the web browser, first the program is to be compiled in the same way as the Java application program as shown below:

```
C:\>Javac SampleApplet.java
```

8.4.1 Running the applet using AppletViewer

AppletViewer is a program which provides a Java run-time environment for applets. It accepts a HTML file as the input and executes the <applet> reference, ignoring the HTML statements. Usually, AppletViewer is used to test Java applets.

To execute any applet program using AppletViewer the applet tag should be added in the comment lines of the program. These comment lines will not be considered when the programs are compiled. The command AppletViewer <appletfile.java> is used to view the applet. To execute Program 8.1, we can invoke the same as follows:

```
c:\>appletviewer SampleApplet.java
```

The reader can thus execute Program 8.1. It is suggested as an exercise.

The drawstring() method draws the String 'This is a sample applet' on the applet panel,starting at the coordinate (50, 50). Note that the applet panel provides a graphical environment in which the drawing will paint rather than print. Usually, the panel is specified in terms of *pixels* in which the upper left corner is the origin (0, 0).

As mentioned earlier, applets interact with the user through AWT. In Program 8.1 we have used one AWT component—button. Observe the SampleApplet class header:

```
public class SampleApplet extends Applet implements ActionListener
```

The ActionListener interface is implemented by a class to handle the events generated by components such as buttons. The method actionPerformed() available in this interface is overridden so that some action is performed when the button is pressed. In Program 8.1, it can be observed that when the applet is first executed, the paint() method does not display anything on the applet window. This is because flag is set to false. When the button is pressed (observe the actionPerformed() method) the flag is set to true and repaint() method is called. This time since the flag is set to true, the repaint() method will display the string 'This is a simple applet'.

In Program 8.1 we use the System.out.println() method to print the sequence of the methods executed when the applet is running.

The output shown in the console is in the following :

```
init() is called;
start() is called;
paint() is called;
stop() is called;
start() is called;
paint() is called;
stop() is called;
destroy() is called;
```

When the applet program is first executed, three methods—init(), start(), paint() are executed. Later, if the applet window is minimized, the stop() method is executed. If the applet window is

maximized again, the start() and the paint() methods will be executed simultaneously. When the applet window is closed, the stop() and the destroy() methods will be called simultaneously.

8.4.2 Running the applet using the web browser

The browsers that support applets include Netscape Navigator, Internet Explorer and Hot Java. The applet can be viewed from a web browser by writing the APPLET tag within the HTML code. To the run Program 8.1 in the web browser, the following html code has to be written:

```
<html>
<head>
<title> A sample Applet</title >
</head>
<body>
<applet code="SampleApplet" width=200 height=200>
</applet>
</body>
</html>
```

To run the applet, the browser, which could be the Internet Explorer for instance, has to be opened, the options File → Open have to be selected. Then, the applet code HTML file is opened by using the Browse button.

To show the difference between running the applet in the applet viewer and the web browser Program 8.1 is modified a little to the form shown in Program 8.2:

Program 8.2 An alternate way of running an applet.

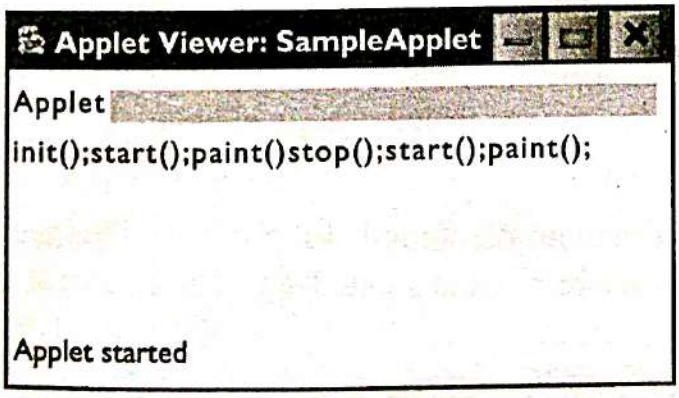
```
import java.applet.*;
import java.awt.*;
public class SampleApplet extends Applet
{
    String msg = "";
    public void init()
    {
        msg = msg+"init();";
    }
    public void start()
    {
        msg = msg+"start();";
    }
```

```
    public void destroy()
    {
        msg = msg+"destroy();";
    }
    public void stop()
    {
        msg = msg+"stop();";
    }

    public void paint(Graphics g)
    {
        msg = msg+ "paint();";
        g.drawString(msg,10,20);
    }
}
```

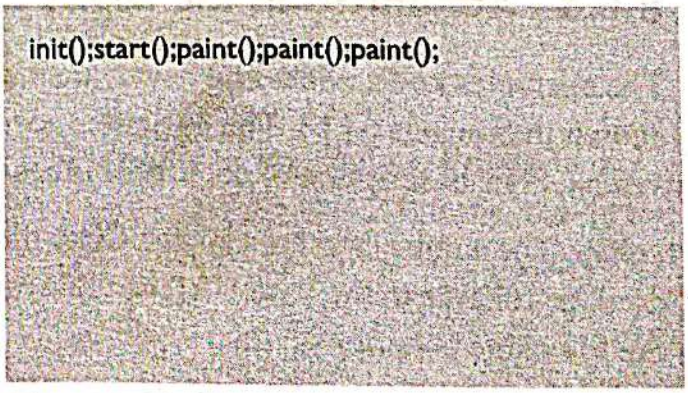
The button is removed and the life-cycle methods are displayed on the applet window instead of on the console. The difference between the output of the above applet when called using the AppletViewer and the web browser is in Figure 8.3.

The output of Program 8.2 shows the applet with the sequence of the methods executed. When the applet program is first executed, the init(), start() and paint() methods are called .

When the web browser is minimized, the paint() method is called, and when it is maximized, paint() is called again. In the case of the applet viewer the stop(), start() and paint() methods are called when the applet is maximized. The same things happen when the applet viewer and the web browser are closed. That is, the stop() method and the destroy() method are called. When moving from the web page containing the applet to another web page and back, the start() method in the applet is called.



(a)



(b)

Fig. 8.3 (a) Output using Applet Viewer; (b) Output using web browser.

8.5 The HTML APPLET Tag

The APPLET tag of HTML is used to start an applet either from a web browser or an applet viewer. The HTML tag allows a Java applet to be embedded in an HTML document.

A number of optional attributes can be used to control an applet's appearance, for example, the dimensions of the applet panel. Also, parameters can be independently defined which can be passed to the applet to control its behaviour.

The complete syntax of the applet tag is given below:

```
<applet
    code = "appletFile" -or- [object = "serializedApplet"]
    width = "pixels"
    height = "pixels"  [codebase = "codebaseURL"]
    [archive = "archiveList"]
    [alt = "alternateText"]
    [name = "appletInstanceName"]
    [align = "alignment"]
    [vspace = "pixels"]
    [hspace = "pixels"]
>
[<param name = "appletAttribute1" value = "value">]
[<param name = "appletAttribute2" value = "value">]
....
....
[alternateHTML]
</applet>
```

8.5.1 Attributes in the applet tag.

The attributes code, width and height are mandatory and all other attributes are options, which are shown in brackets ([ ]). The attributes need not follow the order that was specified in the above syntax. The description of each attribute is given below:

code The name of the Java applet. Class file, which contains the compiled applet code. This name must be relative to the base URL of the Applet and cannot be an absolute URL. The extension in the file name is optional.

width This refers to the width of the applet panel specified in pixels in the browser window.

height This refers to the height of the applet panel specified in pixels in the browser window. The width and height attributes specify the applet's, display area. This applet area does not include any windows or dialogue boxes that the applet shows.

codebase This refers to the base URL of the applet. That is, the URL of the directory that contains the applet class file. If this attribute is not specified then the HTML document's URL directory is taken as the CODEBASE.

archive There are one or more archive files containing Java classes and other resources that will be preloaded. The classes are loaded using an instance of the AppletClassLoader class with the given codebase. The archives in archivelist are separated by a comma (,).

alt This refers to text to be displayed if the browser understands the applet tag but cannot run Java applets.

name This refers to a name for the applet instance which makes it possible for applets to communicate with each other on the same HTML page. The getApplet() method, which is defined in the AppletContext interface, can be used to get the applet instance using name.

align This refers to the alignment of the applet. The possible attribute values are LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE and ABSBOTTOM.

vspace This refers to the space, specified in number of pixels, used as a margin above and below the applet.

hspace This refers to the space, specified in number of pixels, used as a margin to the left and right of the applet.

param The param tag, specifies an applet parameter as a name–value pair. The name is the parameters name, value is its value. The values of these parameters can be obtained using the getParameter() method.

alternateHTML This is ordinary HTML to be displayed in the absence of Java or if the browser does not understand the applet tag. This is different from the alt attribute which understands the applet tag but cannot run, whereas the alternateHTML tag is provided when a browser does not support applets.

8.5.2 Passing parameters to applets

The applet tag helps to define the parameters that are to be passed to the applets. Programs 8.3 and 8.4 illustrate the method of passing parameters to an applef. In this example the first parameter that is presented contains text whose location is specified by the other two parameters. Program 8.3 is the HTML code for this.

Program 8.3 HTML code for the applet program for passing parameters.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Example for Parameter passing to applet </TITLE>
```

```
</HEAD>
<BODY>
<H3>This is an example for passing parameters to Applet</H3><p>
The parameters are<p>
text= Text Parameter<p>
x=50<p>
y=50<p>
<H5> THE TEXT PARAMETER WILL BE DISPLAYED IN THE APPLET AT THE
                                                X, Y LOCATION</H5>
<APPLET CODE="ParamApplet.class" WIDTH = 300 HEIGHT=100 >
<PARAM NAME="text" VALUE="Text Parameter">
<PARAM NAME="x" VALUE="50">
<PARAM NAME="y" VALUE="50">
</APPLET>
</BODY>
</HTML>
```

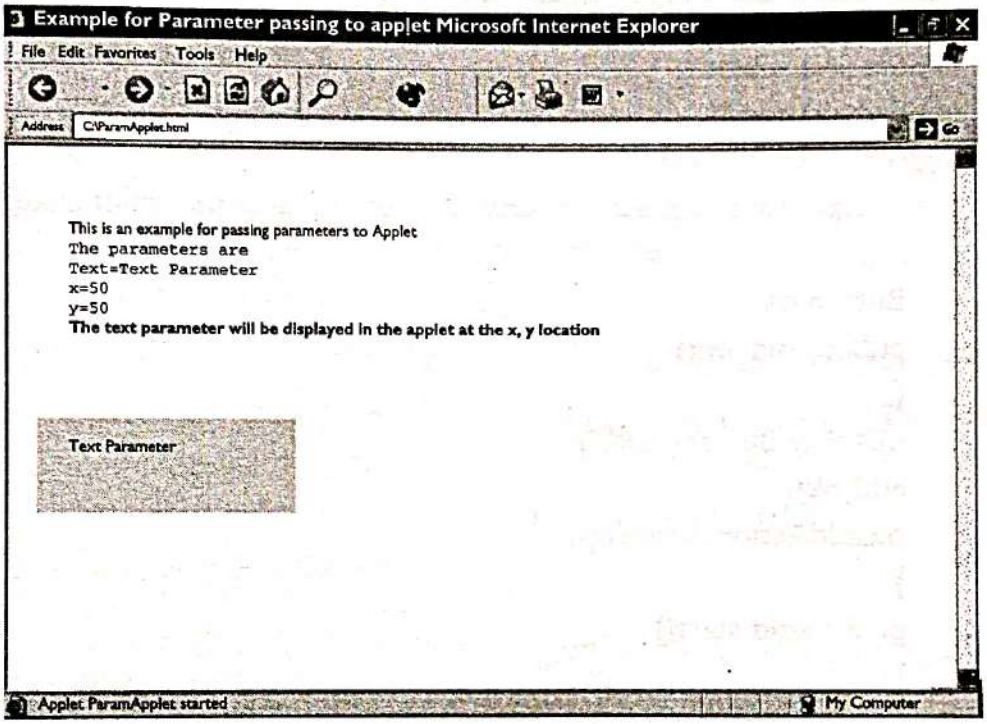
After writing the HTML code let us write the applet program that will retrieve the parameters specified in the HTML tag by using the **getParameter()** method. This method has one argument, which specifies the name of the argument and returns the value of the argument in the form of a string. The **Integer.parseInt()** method is used to convert the parameters to the integer datatype from the string datatype.

Program 8.4 Passing parameters to an applet.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
public class Para Applet extends Applet
{
    public void init()
    {
    }
    public void start()
      {}
    public void destroy()
      {}
    public void stop()
      {}
```

```
    public void paint(Graphics g)
    {
        String text=getParameter("text");
        g.drawString(text,50,50);
    }
}
```

The output of Program 8.4 is the following:



8.6 The java.Applet package

The java.Applet package is one of the smallest packages in the Java API. It contains a single class, the Applet class, and three interfaces: AppletContext, AppletStub and AudioClip. Applets use many GUI components such as labels, checkboxes, buttons, radio buttons, lists, text components, canvases and scroll bars defined in the java.awt package.

The Applet class The Applet class contains a single default constructor with no parameters. This constructor is generally not used. Instead applets are constructed by the run-time environment when they are loaded and do not have to be explicitly constructed.

(See http://www.universitiespress.com/downloads/books/81-7371-572-6.pdf, Table 8.3 for a description of the methods in the Applet class.)

The java.applet.Applet class is actually a sub-class of java.awt.Panel. Thus, an applet can be used directly as the starting point for an AWT layout. Since an applet itself is a Panel, by default it has a Flow Layout manager. The Applet class inherits the methods of the Component, Container and Panel classes.

The AppletContext interface To get information from the applet's execution environment, Java provides an interface called AppletContext. The following are the methods of AppletContext. Within an applet, if its context is obtained then we can bring other documents into view by calling the showDocument() methods.The methods getDocumentBase() and getCodeBase() are used to get the URL objects of the HTML document and the applet class file. They can be concatenated with a string that names the file you want to load. Program 8.5 illustrates the use of the methods in the Applet Context interface.

Program 8.5 Methods in the Applet Context interface.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class ParamApplet1 extends Applet implements ActionListener
{
    Button ok;
    public void init()
    {
    ok=new Button("OK");
    add(ok);
    ok.addActionListener();
    }
    public void start()
    {
    }
    public void destroy()
    {
    }
    public void stop()
    {
    }
    public void paint(Graphics g)
    {
    }
    public void actionPerformed(ActionEvent e )
    {
       if(e.getActionCommand().equals("OK"))
       {
          String str=getDocumentBase();
          try{
                URL url=new URL(str, "samplehtml2.html");
```

```
                    getAppletContext().showDocument(url);
                }catch(MalformedURLException e)
                {
                }
            }
        }
    }
```

Table 8.4 gives a list of the methods in this interface and describes briefly how they can be used.

(See http://www.universitiespress.com/downloads/books/81-7371-572-6.pdf, Table 8.4 for a description of the methods in the AppletContext interface.)

The AppletStub interface The AppletStub interface provides a way to get information from the run-time browser environment.

(See http://www.universitiespress.com/downloads/books/81-7371-572-6.pdf, Table 8.5 for a description of the methods in this interface.)

The AudioClip interface To load audio clip into the applet the method **getAudioclip()** is called.

(See http://www.universitiespress.com/downloads/books/81-7371-572-6.pdf, Table 8.6 for a description of methods in the AudioClip interface.)

8.7 Sample Programs

1. Program 8.6 shows the basic functionalities (**init()**, **start()** and **paint()**) of an applet. It also illustrates the steps involved in processing the applet.

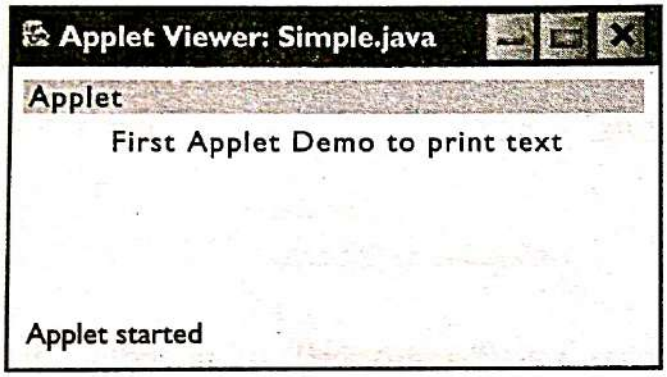
Program 8.6 A simple applet program.

```
import java.applet.Applet;
import java.awt.Graphics;

public class Simple extends Applet
{
    public void init()
    {
        System.out.println("Initializing...");
    }

    public void start()
    {
        System.out.println("starting... ");
    }
```

```
    public void stop()
    {
        System.out.println("stopping... ");
    }
    public void paint(java.awt.Graphics g)
    {
        g.drawString("First Applet Demo to print text",50,25);
        System.out.println("First Applet Demo to print text");
    }
}
/*<applet code="Simple.java" width=300 height=200></applet>*/
```

The output of Program 8.6 is shown below.

Applet Window



Console Output

```
E:\>javac Simple.java
E:\>AppletViewer Simple.java
Initializing...
starting...
First Applet Demo to print text
First Applet Demo to print text
stopping...
E:\>
```

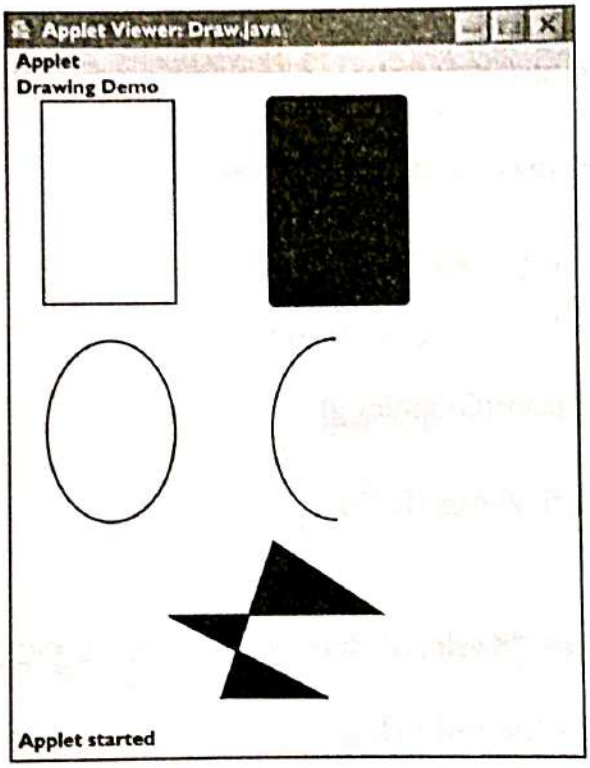
2. Program 8.7 shows the basic functionalities of the graphics class inherited from the java.awt package.

Program 8.7 The basic functionalities of the Graphics class.

```
//This program is to illustrate the basic functionalities of Graphics class inherited
from awt package. This program prints out the drawings in an applet window.
import java.awt.Graphics;
import java.applet.Applet;
public class Draw  extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("Drawing Demo",10,10);
        g.drawRect(35, 20, 125, 200);
```

```
        g.fillRoundRect(250, 20, 125, 200, 15, 15);
        g.drawOval(35, 250, 125, 180);
        g.drawArc(250, 250, 125, 180, 90, 180);
        int x[] = {250, 350, 150, 300, 200};
        int y[] = {450, 520, 520, 600, 600};
        int numPts = 5;
        g.drawPolygon(x, y, numPts);
        g.fillPolygon(x, y, numPts);
    }
}
/*<applet code="Draw.java" width=500 height=500 > </applet> */
```

The output of Program 8.7 is shown below:



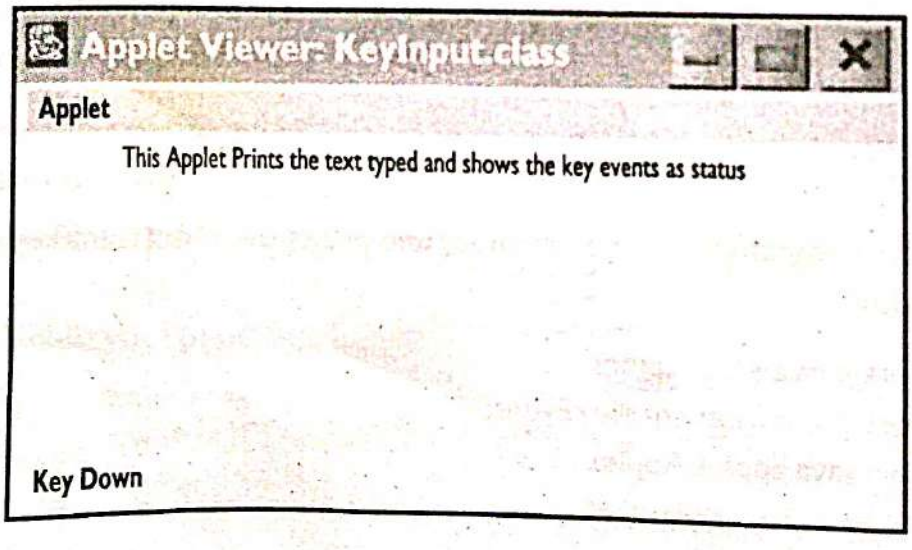
3. Program 8.8 illustrates the keyboard events of applets

Program 8.8 Keyboard events of applets.

```
// This program takes the key input and prints the characters keyed in on the applet
window.

import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.applet.Applet;
```

```
public class  KeyInput extends Applet implements KeyListener
{
    String msg;
    public void init()
    {
        addKeyListener(this);
        requestFocus();
        msg= "  ";
    }
    public void keyPressed(KeyEvent ke)
    {
        showStatus("Key Down");
    }
    public void keyReleased(KeyEvent ke)
    {
        showStatus("Key Released");
    }
    public void keyTyped(KeyEvent ke)
    {
        msg +=ke.getKeyChar();
        repaint();
    }
    public void paint(Graphics g)
    {
        g.drawString(msg,10,10);
    }
}
/* <applet  code= "KeyInput.class" width=300 height=300 > </applet> */
```

The output of Program 8.8 is the following:

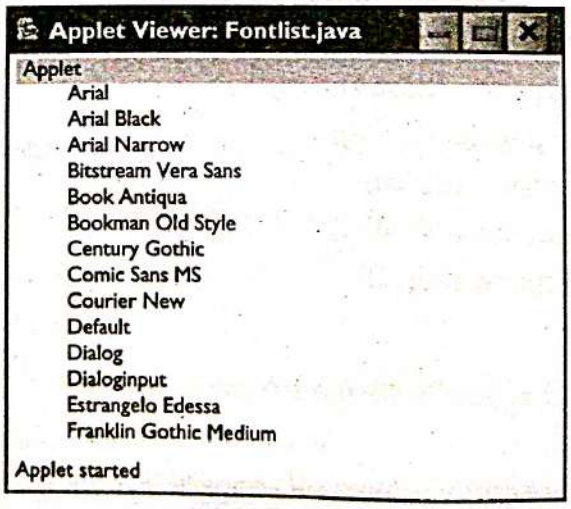


4. Program 8.9 uses the available fonts in the system and prints them on the applet.

Program 8.9 Printing the fonts available in the system.

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.GraphicsEnvironment;
import java.applet.Applet;

public class FontList extends Applet
{
    public void paint(Graphics g)
    {
        Font f;
        String s;
        GraphicsEnvironment ge =
        GraphicsEnvironment.getLocalGraphicsEnvironment();

        String fontNames[ ] =
            ge.getAvailableFontFamilyNames();
        g.drawString("Hello",0,0);

        for(int i=0,j=10; i<fontNames.length; i++,j+=20)
        {
            g.drawString(fontNames[i],50,j);
            System.out.println( fontNames[i] );
        }
    }
}
/*<applet code="FontList.java" width=300 height=200 > </applet> */
```

The output of Program 8.9 is as shown below:



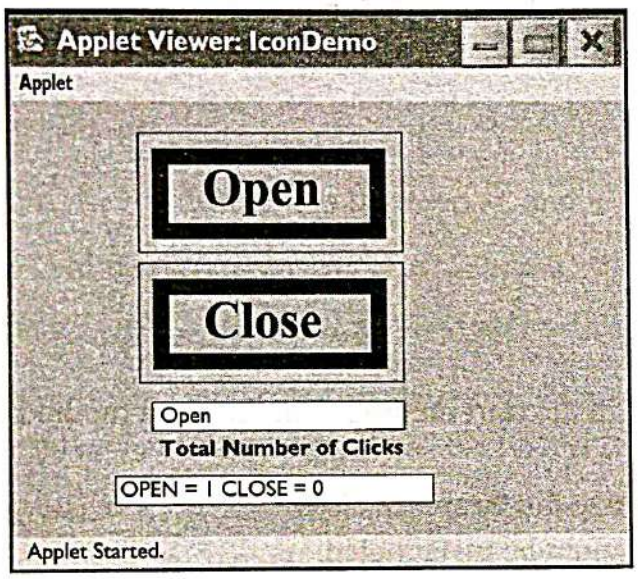
5. Program 8.10 demonstrates icons in an applet and mouse action event-handling.

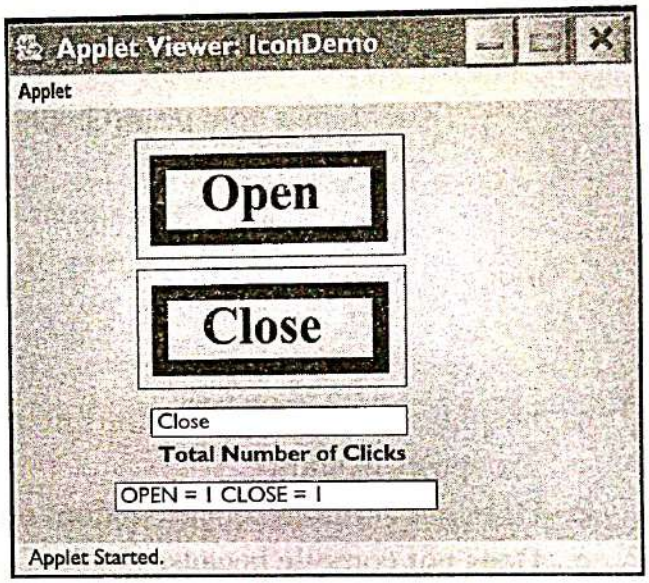
Program 8.10 Using icons and mouse action event-handling.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
/*<applet code="IconDemo" width=300 height=400></applet>*/
public class IconDemo extends JApplet implements ActionListener
{
    JButton ob1,ob2;
    JLabel label;
    JTextField tf,tf2;
    static int flag1;
    static int flag2;
    public void init()
    {
        Container contentpane= getContentPane();
        contentpane.setLayout(new FlowLayout());
        ImageIcon ic1=new ImageIcon("open.gif","Java Logo");
        ob1=new JButton(ic1);
        ob1.setActionCommand("Open");
        ob1.addActionListener(this);

        ImageIcon ic2=new ImageIcon("close.gif");
        ob2=new JButton(ic2);
        ob2.setActionCommand("Close");
        ob2.addActionListener(this);

        tf= new JTextField(15);
        tf2= new JTextField(20);
        label= new JLabel("Total Number of Clicks");
        contentpane.add(ob1);
        contentpane.add(ob2);
        contentpane.add(tf);
        contentpane.add(label);
        contentpane.add(tf2);
    }

    public void actionPerformed(ActionEvent ae)
    {
        if(ae.getActionCommand().equals("Open"))
```

```
            flag1++;
        else if(ae.getActionCommand().equals("Close"))
            flag2++;
        tf.setText(ae.getActionCommand());
        tf2.setText("OPEN ="+flag1+ "CLOSE =" +flag2);
    }
}
```

The output of Program 8.11 is shown below:





Objective type questions

1. The object of _____ type is passed as a parameter to the paint method.
2. The _____ method of applet is called first when the applet is minimized and again restored.
3. The _____ method of an applet is called only once in an applet's life-cycle.
4. The showDocument() method with a single parameter of the AppletContext interface takes objects of _____ type as its argument.
   a. Graphics b. URL c. String d. Object
5. What happens when the following code is executed with AppletViewer?

```
import java.applet.*;
import java.io.*;
import java.awt.*;
public class AppletEg extends Applet
```