Course: Object Based Modeling
Code: CS-33105
Branch: MCA-3

Lecture 15: Input/Output (I/O): Exploring java.io

Dr. J Sathish Kumar (JSK)
(Faculty & Coordinator)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Introduction

- As all programmers learn early on, most programs cannot accomplish their goals without accessing external data.
- Data is retrieved from an *input* source.
- The results of a program are sent to an *output* destination.
- The **io** package supports Java's basic I/O (input/output) system, including file I/O.
- For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes.
- Although physically different, these devices are all handled by the same abstraction: the *stream*

The I/O Classes

BufferedInputStream	FileWriter	PipedOutputStream
BufferedOutputStream	FilterInputStream	PipedReader
BufferedReader	FilterOutputStream	PipedWriter
BufferedWriter	FilterReader	PrintStream
ByteArrayInputStream	FilterWriter	PrintWriter
ByteArrayOutputStream	InputStream	PushbackInputStream
CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile
Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

The I/O Interfaces

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

Object serialization (the storage and retrieval of objects)

File

- Although most of the classes defined by **java.io** operate on streams, the **File** class does not.
- 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.
- Files are a primary source and destination for data within many programs.
- A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.
- The following constructors can be used to create **File** objects:
  File(String directoryPath)
  File(String directoryPath, String filename)
  File(File dirObj, String filename)
  File(URI uriObj)

File

- The following example creates three files: **f1**, **f2**, and **f3**.
- The first **File** object is constructed with a directory path as the only argument.
- The second includes two arguments—the path and the filename.
- The third includes the file path assigned to **f1** and a filename; **f3** refers to the
  same file as **f2**.

```
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

Example #1

```
// Demonstrate File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");

    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

This program will produce output similar to this:

```
File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1282832030047
File size: 695 Bytes
```

File methods

Method	Description
void deleteOnExit( )	Removes the file associated with the invoking object when the Java Virtual Machine terminates.
long getFreeSpace( )	Returns the number of free bytes of storage available on the partition associated with the invoking object.
long getTotalSpace( )	Returns the storage capacity of the partition associated with the invoking object.
long getUsableSpace( )	Returns the number of usable free bytes of storage available on the partition associated with the invoking object.
boolean isHidden( )	Returns **true** if the invoking file is hidden. Returns **false** otherwise.
boolean setLastModified(long *millisec*)	Sets the time stamp on the invoking file to that specified by *millisec*, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC).
boolean setReadOnly( )	Sets the invoking file to read-only.

Directories Example #2

```
// Using directories.
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);

    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```

sample output from the program.

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory
```

Using FilenameFilter

- You will often want to limit the number of files returned by the **list( )** method to include only those files that match a certain filename pattern, or *filter*.
- To do this, you must use a second form of **list( )**, shown here:
  String[ ] list(FilenameFilter *FFObj*)
- In this form, *FFObj* is an object of a class that implements the **FilenameFilter** interface.
- **FilenameFilter** defines only a single method, **accept( )**, which is called once for each file in a list.
- Its general form is given here:
  boolean accept(File *directory*, String *filename*)

Example #3

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

```
// Directory of .HTML files.
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

I/O Exceptions

- Two exceptions play an important role in I/O handling. The first is **IOException**.
- As it relates to most of the I/O classes described, if an I/O error occurs, an **IOException** is thrown.
- In many cases, if a file cannot be opened, a **FileNotFoundException** is thrown.
- **FileNotFoundException** is a subclass of **IOException**, so both can be caught with a single **catch** that catches **IOException**.
- Another exception class that is sometimes important when performing I/O is **SecurityException**.

The Stream Classes

- Java's stream-based I/O is built upon four abstract classes:
  - **InputStream**,
  - **OutputStream**,
  - **Reader**, and
  - **Writer**.
- **InputStream** and **OutputStream** are designed for byte streams.
- **Reader** and **Writer** are designed for character streams.
- The byte stream classes and the character stream classes form separate hierarchies.
- In general, you should use the character stream classes when working with characters or strings and use the byte stream classes when working with bytes or other binary objects.

The Byte Streams

- The byte stream classes provide a rich environment for handling byte-oriented I/O.
- A byte stream can be used with any type of object, including binary data.
- This versatility makes byte streams important to many types of programs.
- Since the byte stream classes are topped by **InputStream** and **OutputStream**

InputStream & OutputStream

- **InputStream** is an abstract class that defines Java's model of streaming byte input.
- It implements the **AutoCloseable** and **Closeable** interfaces.
- Most of the methods in this class will throw an **IOException** when an I/O error occurs.

- **OutputStream** is an abstract class that defines streaming byte output.
- It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces.
- Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors.

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.

The Methods Defined by InputStream

The 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 an expression without having to cast it 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*].

FileInputStream

- The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file.
- Two commonly used constructors are shown here:

  FileInputStream(String filePath)
  FileInputStream(File fileObj)
- Either can throw a **FileNotFoundException**.
- Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.
- The following example creates two **FileInputStream**s 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 #4

```
// Demonstrate FileInputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileInputStreamDemo {
  public static void main(String args[]) {
    int size;

    // Use try-with-resources to close the stream.
    try ( FileInputStream f =
           new FileInputStream("FileInputStreamDemo.java") ) {

      System.out.println("Total Available Bytes: " +
                         (size = f.available()));

      int n = size/40;
      System.out.println("First " + n +
                         " bytes of the file one read() at a time");
      for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
      }

      System.out.println("\nStill Available: " + f.available());

      System.out.println("Reading the next " + n +
                         " with one read(b[])");
      byte b[] = new byte[n];
      if (f.read(b) != n) {
        System.err.println("couldn't read " + n + " bytes.");
      }
```

```
      System.out.println(new String(b, 0, n));
      System.out.println("\nStill Available: " + (size = f.available()));
      System.out.println("Skipping half of remaining bytes with skip()");
      f.skip(size/2);
      System.out.println("Still Available: " + f.available());

      System.out.println("Reading " + n/2 + " into the end of array");
      if (f.read(b, n/2, n/2) != n/2) {
        System.err.println("couldn't read " + n/2 + " bytes.");
      }

      System.out.println(new String(b, 0, b.length));
      System.out.println("\nStill Available: " + f.available());
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1785
First 44 bytes of the file one read() at a time
// Demonstrate FileInputStream.
// This pr
Still Available: 1741
Reading the next 44 with one read(b[])
ogram uses try-with-resources. It requires J

Still Available: 1697
Skipping half of remaining bytes with skip()
Still Available: 849
Reading 22 into the end of array
ogram uses try-with-rebyte[n];
      if (

Still Available: 827
```

FileOutputStream

- **FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file.
- It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces.
- Four of its constructors are shown here:
  FileOutputStream(String filePath)
  FileOutputStream(File fileObj)
  FileOutputStream(String filePath, boolean append)
  FileOutputStream(File fileObj, boolean append)
- They can throw a **FileNotFoundException**.
- Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.
- If *append* is **true**, the file is opened in append mode.

Example #5

```
// Demonstrate FileOutputStream.
// This program uses the traditional approach to closing a file.

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";

      byte buf[] = source.getBytes();
      FileOutputStream f0 = null;
      FileOutputStream f1 = null;
      FileOutputStream f2 = null;

      try {
        f0 = new FileOutputStream("file1.txt");
        f1 = new FileOutputStream("file2.txt");
        f2 = new FileOutputStream("file3.txt");

        // write to first file
        for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

        // write to second file
        f1.write(buf);

        // write to third file
        f2.write(buf, buf.length-buf.length/4, buf.length/4);
      } catch(IOException e) {
        System.out.println("An I/O Error Occurred");
```

```
    } finally {
      try {
        if(f0 != null) f0.close();
      } catch(IOException e) {
        System.out.println("Error Closing file1.txt");
      }
      try {
        if(f1 != null) f1.close();
      } catch(IOException e) {
        System.out.println("Error Closing file2.txt");
      }
      try {
        if(f2 != null) f2.close();
      } catch(IOException e) {
        System.out.println("Error Closing file3.txt");
      }
    }
  }
}
```

Here are the contents of each file after running this program. First, **file1.txt**:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Next, **file2.txt**:

```
Now is the time for all good men
 to come to the aid of their country
 and pay their due taxes.
```

Finally, **file3.txt**:

```
nd pay their due taxes.
```

Example #6

```
// Demonstrate FileOutputStream.
// This version uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();

    // Use try-with-resources to close the files.
    try (FileOutputStream f0 = new FileOutputStream("file1.txt");
         FileOutputStream f1 = new FileOutputStream("file2.txt");
         FileOutputStream f2 = new FileOutputStream("file3.txt") )
    {

      // write to first file
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

      // write to second file
      f1.write(buf);

      // write to third file
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    }
  }
}
```

ByteArrayInputStream

- **ByteArrayInputStream** is an implementation of an input stream that uses a byte array as the source.
- This class has two constructors, each of which requires a byte array to provide the data source:
  ByteArrayInputStream(byte array [ ])
  ByteArrayInputStream(byte array [ ], int start, int numBytes)
- Here, *array* is the input source.
- The second constructor creates an **InputStream** from a subset of the byte array that begins with the character at the index specified by *start* and is *numBytes* long.

Example #7

```
import java.io.*;

class ByteArrayInputStreamReset {
  public static void main(String args[]) {
    String tmp = "abc";
    byte b[] = tmp.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(b);

    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read()) != -1) {
        if (i == 0) {
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
      System.out.println();
      in.reset();
    }
  }
}
```

Here's the output:

```
abc
ABC
```

ByteArrayOutputStream

- **ByteArrayOutputStream** is an implementation of an output stream that uses a byte array as the destination.
- **ByteArrayOutputStream** has two constructors, shown here:
  ByteArrayOutputStream( )
  ByteArrayOutputStream(int numBytes)

Example #8

```
// Demonstrate ByteArrayOutputStream.
// This program uses try-with-resources. It requires JDK 7 or later

import java.io.*;

class ByteArrayOutputStreamDemo {
  public static void main(String args[]) {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "This should end up in the array";
    byte buf[] = s.getBytes();

    try {
      f.write(buf);
    } catch(IOException e) {
      System.out.println("Error Writing to Buffer");
      return;
    }

    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");
    byte b[] = f.toByteArray();
    for (int i=0; i<b.length; i++) System.out.print((char) b[i]);

    System.out.println("\nTo an OutputStream()");
```

```
// Use try-with-resources to manage the file stream.
try ( FileOutputStream f2 = new FileOutputStream("test.txt") )
{
  f.writeTo(f2);
} catch(IOException e) {
  System.out.println("I/O Error: " + e);
  return;
}
```

```
      System.out.println("Doing a reset");
      f.reset();

      for (int i=0; i\<3; i++) f.write('X');

      System.out.println(f.toString());
    }
  }
```

Here's the output:

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

Buffered Byte Streams

- For the byte-oriented streams, a *buffered stream* extends a filtered stream class by attaching a memory buffer to the I/O stream.
- This buffer allows Java to do I/O operations on more than a byte at a time, thereby improving performance.
- Because the buffer is available, skipping, marking, and resetting of the stream become possible.
- The buffered byte stream classes are **BufferedInputStream** and **BufferedOutputStream**.

BufferedInputStream

- Buffering I/O is a very common performance optimization.
- Java's **BufferedInputStream** class allows you to "wrap" any **InputStream** into a buffered stream to improve performance.
- **BufferedInputStream** has two constructors:
  BufferedInputStream(InputStream inputStream)
  BufferedInputStream(InputStream inputStream, int bufSize)

Example #9

```
// Use buffered input.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    byte buf[] = s.getBytes();

    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    int c;
    boolean marked = false;

    // Use try-with-resources to manage the file.
    try ( BufferedInputStream f = new BufferedInputStream(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '&':
          if (!marked) {
            f.mark(32);
            marked = true;
          } else {
            marked = false;
          }
          break;
        case ';':
          if (marked) {
            marked = false;
            System.out.print("(c)");
          } else
            System.out.print((char) c);
          break;
        case ' ':
          if (marked) {
            marked = false;
            f.reset();
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if (!marked)
            System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

```
This is a (c) copyright symbol but this is &copy not.
```

BufferedOutputStream

- A **BufferedOutputStream** is similar to any **OutputStream** with the exception that the **flush( )** method is used to ensure that data buffers are written to the stream being buffered.
- Since the point of a **BufferedOutputStream** is to improve performance by reducing the number of times the system actually writes data, you may need to call **flush( )** to cause any data that is in the buffer to be immediately written.
- Here are the two available constructors:
  BufferedOutputStream(OutputStream outputStream)
  BufferedOutputStream(OutputStream outputStream, int bufSize)

PrintStream

- The **PrintStream** class provides all of the output capabilities we have been using from the **System** file handle, **System.out**.
- This makes **PrintStream** one of Java's most often used classes.
- It implements the **Appendable**, **AutoCloseable**,**Closeable**, and **Flushable** interfaces.
- **PrintStream** defines several constructors.
- The ones shown next have been specified from the start:
  PrintStream(OutputStream outputStream)
  PrintStream(OutputStream outputStream, boolean autoFlushingOn)
  PrintStream(OutputStream outputStream, boolean autoFlushingOn String charSet)
  throws UnsupportedEncodingException

PrintStream

- The next set of constructors gives you an easy way to construct a **PrintStream** that writes its output to a file:

  PrintStream(File outputFile) throws FileNotFoundException
  PrintStream(File outputFile, String charSet)
  throws FileNotFoundException, UnsupportedEncodingException
  PrintStream(String outputFileName) throws
  FileNotFoundException
  PrintStream(String outputFileName, String charSet) throws
  FileNotFoundException,UnsupportedEncodingException

Example #10

```
// Demonstrate printf().

class PrintfDemo {
  public static void main(String args[]) {
    System.out.println("Here are some numeric values " +
                       "in different formats.\n");

    System.out.printf("Various integer formats: ");
    System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

    System.out.println();
    System.out.printf("Default floating-point format: %f\n",
                      1234567.123);
    System.out.printf("Floating-point with commas: %,f\n",
                      1234567.123);

    System.out.printf("Negative floating-point default: %,f\n",
                      -1234567.123);
    System.out.printf("Negative floating-point option: %,(f\n",
                      -1234567.123);

    System.out.println();

    System.out.printf("Line up positive and negative values:\n");
    System.out.printf("% ,.2f\n% ,.2f\n",
                      1234567.123, -1234567.123);
  }
}
```

The output is shown here:

```
Here are some numeric values in different formats.

Various integer formats: 3 (3) +3 00003

Default floating-point format: 1234567.123000
Floating-point with commas: 1,234,567.123000
Negative floating-point default: -1,234,567.123000
Negative floating-point option: (1,234,567.123000)

Line up positive and negative values:
  1,234,567.12
 -1,234,567.12
```

Other Byte Streams

- Filtered Byte Streams
  - FilterOutputStream(OutputStream *os*)
    FilterInputStream(InputStream *is*)
- SequenceInputStream
  - SequenceInputStream(InputStream *first*, InputStream *second*)
    SequenceInputStream(Enumeration <? extends InputStream> *streamEnum*)
- DataOutputStream and DataInputStream
  - DataOutputStream(OutputStream *outputStream*)
- RandomAccessFile
  - RandomAccessFile(File *fileObj*, String *access*) throws FileNotFoundException
  - RandomAccessFile(String *filename*, String *access*) throws FileNotFoundException

The Character Streams

- While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters.
- Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters.
- Among, several of the character I/O classes, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.

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.
int read(CharBuffer *buffer*)	Attempts to read 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.

The Methods Defined by Reader

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 output 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 an expression without having to cast it back to **char**. However, only the low-order 16 bits are written.
void write(char *buffer*[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*.

The Methods Defined by Writer

FileReader

- The **FileReader** class creates a **Reader** that you can use to read the contents of a file.
- Two commonly used constructors are shown here:
  FileReader(String *filePath*)
  FileReader(File *fileObj*)
- Either can throw a **FileNotFoundException**.
- Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

```
// Demonstrate FileReader.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

    try ( FileReader fr = new FileReader("FileReaderDemo.java") )
    {
      int c;

      // Read and display the file.
      while((c = fr.read()) != -1) System.out.print((char) c);

    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Example #11

FileWriter

- **FileWriter** creates a **Writer** that you can use to write to a file.
- Four commonly used constructors are shown here:
  FileWriter(String *filePath*)
  FileWriter(String *filePath*, boolean *append*)
  FileWriter(File *fileObj*)
  FileWriter(File *fileObj*, boolean *append*)
- They can all throw an **IOException**.
- Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.
- If *append* is **true**, then output is appended to the end of the file.

```
// Demonstrate FileWriter.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileWriterDemo {
  public static void main(String args[]) throws IOException {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    char buffer[] = new char[source.length()];
    source.getChars(0, source.length(), buffer, 0);

    try ( FileWriter f0 = new FileWriter("file1.txt");
          FileWriter f1 = new FileWriter("file2.txt");
          FileWriter f2 = new FileWriter("file3.txt") )
    {
      // write to first file
      for (int i=0; i < buffer.length; i += 2) {
        f0.write(buffer[i]);
      }

      // write to second file
      f1.write(buffer);

      // write to third file
      f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);

    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    }
  }
}
```

Example #12

Other Character Streams

- CharArrayReader
  - CharArrayReader(char *array* [ ])
    CharArrayReader(char *array* [ ], int *start*, int *numChars*)
- CharArrayWriter
  - CharArrayWriter( )
    CharArrayWriter(int *numChars*)
- BufferedReader
  - BufferedReader(Reader *inputStream*)
    BufferedReader(Reader *inputStream*, int *bufSize*)
- PushbackReader
  - void unread(int *ch*) throws IOException
    void unread(char *buffer* [ ]) throws IOException
    void unread(char *buffer* [ ], int *offset*, int *numChars*) throws IOException

Other Character Streams

- PrintWriter
  - PrintWriter(OutputStream *outputStream*)
    PrintWriter(OutputStream *outputStream*, boolean *autoFlushingOn*)
    PrintWriter(Writer *outputStream*)
    PrintWriter(Writer *outputStream*, boolean *autoFlushingOn*)
- The next set of constructors gives you an easy way to construct a **PrintWriter** that writes its output to a file.
  - PrintWriter(File *outputFile*) throws FileNotFoundException
    PrintWriter(File *outputFile*, String *charSet*)
    throws FileNotFoundException, UnsupportedEncodingException
    PrintWriter(String *outputFileName*) throws FileNotFoundException
    PrintWriter(String *outputFileName*, String *charSet*)
    throws FileNotFoundException, UnsupportedEncodingException

Externalizable

- The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically.
- However, there are cases in which the programmer may need to have control over these processes.
- For example, it may be desirable to use compression or encryption techniques.
- The **Externalizable** interface is designed for these situations.
- The **Externalizable** interface defines these two methods:
  void readExternal(ObjectInput *inStream*)
  throws IOException, ClassNotFoundException
  void writeExternal(ObjectOutput *outStream*)
  throws IOException
- In these methods, *inStream* is the byte stream from which the object is to be read, and *outStream* is the byte stream to which the object is to be written.

Serializable

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

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*.
void writeObject(Object *obj*)	Writes object *obj* to the invoking stream.

ObjectOutputStream

- The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface.
- It is responsible for writing objects to a stream.
- One constructor of this class is shown here:
  ObjectOutputStream(OutputStream *outStream*) throws IOException
- The argument *outStream* is the output stream to which serialized objects will be written.
- Closing an **ObjectOutputStream** automatically closes the underlying stream specified by *outStream*.

Method	Description
void close( )	Closes the invoking stream. Further write attempts will generate an **IOException**. The underlying stream is also closed.
void flush( )	Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers.
void write(byte *buffer*[ ])	Writes an array of bytes to the invoking stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].
void write(int *b*)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBoolean(boolean *b*)	Writes a **boolean** to the invoking stream.
void writeByte(int *b*)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*.
void writeBytes(String *str*)	Writes the bytes representing *str* to the invoking stream.
void writeChar(int *c*)	Writes a **char** to the invoking stream.
void writeChars(String *str*)	Writes the characters in *str* to the invoking stream.
void writeDouble(double *d*)	Writes a **double** to the invoking stream.
void writeFloat(float *f*)	Writes a **float** to the invoking stream.
void writeInt(int *i*)	Writes an **int** to the invoking stream.
void writeLong(long *l*)	Writes a **long** to the invoking stream.
final void writeObject(Object *obj*)	Writes *obj* to the invoking stream.
void writeShort(int *i*)	Writes a **short** to the invoking stream.

Methods Defined by ObjectOutputStream

ObjectInput

- The **ObjectInput** interface extends the **DataInput** and **AutoCloseable** interfaces and defines the methods

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**.
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*, returning the 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 that were successfully read. –1 is returned when the end of the file is encountered.
Object readObject( )	Reads an object from the invoking stream.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored.

ObjectInputStream

- The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface.
- **ObjectInputStream** is responsible for reading objects from a stream.
- One constructor of this class is shown here:
  ObjectInputStream(InputStream inStream) throws IOException
- The argument *inStream* is the input stream from which serialized objects should be read.
- Closing an **ObjectInputStream** automatically closes the underlying stream specified by *inStream*.

Method	Description
int available( )	Returns the number of bytes that are now available in the input buffer.
void close( )	Closes the invoking stream. Further read attempts will generate an **IOException**. The underlying stream is also closed.
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*[ ], 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.
Boolean readBoolean( )	Reads and returns a **boolean** from the invoking stream.
byte readByte( )	Reads and returns a **byte** from the invoking stream.
char readChar( )	Reads and returns a **char** from the invoking stream.
double readDouble( )	Reads and returns a **double** from the invoking stream.
float readFloat( )	Reads and returns a **float** from the invoking stream.
void readFully(byte *buffer*[ ])	Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read.
void readFully(byte *buffer*[ ], int *offset*, int *numBytes*)	Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read.
int readInt( )	Reads and returns an **int** from the invoking stream.
long readLong( )	Reads and returns a **long** from the invoking stream.
final Object readObject( )	Reads and returns an object from the invoking stream.
short readShort( )	Reads and returns a **short** from the invoking stream.
int readUnsignedByte( )	Reads and returns an unsigned **byte** from the invoking stream.
int readUnsignedShort( )	Reads and returns an unsigned **short** from the invoking stream.

Methods Defined by ObjectInputStream

Example #13

```
// A serialization demo.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

    // Object serialization

    try ( ObjectOutputStream objOStrm =
            new ObjectOutputStream(new FileOutputStream("serial")) )
    {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);

      objOStrm.writeObject(object1);
    }
    catch(IOException e) {
      System.out.println("Exception during serialization: " + e);
    }

    // Object deserialization

    try ( ObjectInputStream objIStrm =
            new ObjectInputStream(new FileInputStream("serial")) )
    {
      MyClass object2 = (MyClass)objIStrm.readObject();
      System.out.println("object2: " + object2);
    }
    catch(Exception e) {
      System.out.println("Exception during deserialization: " + e);
    }
  }
}
```

```
class MyClass implements Serializable {
  String s;
  int i;
  double d;

  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }

   public String toString() {
     return "s=" + s + "; i=" + i + "; d=" + d;
   }
 }
```

The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```