

Chapter 8: Input/Output

Contents

Objectives.....	4
Section 1: Introduction.....	5
Java I/O	6
InputStreamReader.....	7
Printing Using the Formatter Class	8
Input Using the Scanner Class	11
BufferedReader	13
Reading an Integer	14
BufferedWriter	15
File Streams.....	16
StreamTokenizer	17
StreamTokenizer Example	18
Section 2: File System Access	19
File Class	20
List Method	21
FilenameFilter	22
Formatting Output.....	23
NumberFormatDemo	24
Section 3: InputStream and OutputStream	25
Input and Output	26
InputStream and OutputStream.....	27
InputStream Methods	28
InputStream Example.....	29
OutputStream Class.....	30
Byte Array Streams	31
File Stream Class.....	32
Piped Streams and Buffered Streams.....	33
Data Streams	34
Data Stream Example.....	35
PrintStream Class	36
RandomAccessFile Class	37
Section 4: New I/O Classes.....	38
New I/O Classes	39
Buffer Class.....	40
Buffer Characteristics.....	41
View Buffer.....	42
put and get Methods	43
Memory-Mapped Files.....	44

Summary45

Objectives

After completing this unit the student will:

- Understand the class hierarchy used for I/O
- Be able to open files for input and output
- Be able to use filter streams to accomplish sophisticated I/O
- Use the File Class to access file system information
- Be able to format numeric output
- Explain the essence of the new I/O Java classes

Section 1: Introduction

Java I/O

Java I/O is based on the concept of streams where a stream is a sequence of bytes. This stream can be either binary or character. The output of one stream is frequently inputted into another stream. Each stream typically does some sort of manipulation of the data or provides additional methods.

This added value is what makes a stream useful. One stream might convert binary data into a character sequence. Another stream may provide random access to a file while another performs a buffering action.

The power of the Java I/O classes is the ability to combine these different types of streams into a sequence that meets the needs of a specific application. Some I/O activities are easier in other languages. The other languages may presume that certain activities will always be performed. Java does not make this presumption and allows the developer to create the functions that they need. The limitation of the I/O is imposed by the ability of the developer, not by the language itself.

There are four major parts of the java.io package that will be covered here:

- Character Streams
- File System Access
- InputStream and OutputStream classes
- New I/O classes

Character streams classes derive from the Reader and Writer classes. Characters streams generally provide better access than those classes derived from InputStream or OutputStream. They should be used instead of classes derived from InputStream and OutputStream whenever possible.

InputStreamReader

There are three predefined streams in Java:

- System.in – InputStream
- System.out – PrintStream
- System.err – PrintStream

System.in is a binary stream. It is normally desirable to convert this stream to characters before it can be used. The InputStreamReader can be used for this purpose.

```
InputStreamReader isr = new InputStreamReader(System.in);
```

This stream is unbuffered.

Printing Using the Formatter Class

The Formatter class is an interpreter that is intended to be used to support C printf style statements in Java. It provides similar capabilities to C's printf but is more restrictive. For example, a mismatched of flags and data type will result in an error being thrown. In C, this condition was silently ignored which often resulted in unexpected behavior and hard to debug code.

The following illustrates the PrintStream printf method:

```
System.out.printf("%1$d divided by %3$d is %2$f\n",
                  num1, result, num2);
```

The string:

`%1$d`

is used to control the formatting of the integer, num1. The general syntax for these specifiers consists of several parts:

`%[argument_index$][flags][width][.precision]conversion`

Brackets, [and], are used to indicate an option part of the specifier. Each part of the specifier is detailed in the following table:

Specifier Field	Optional	Usage
%	No	Indicates the start of a specifier
argument_index\$	Yes	The position of the argument in the printf argument list
flags	Yes	Used to modify the conversion (The argument's value to its string representation)
width	Yes	The minimum number of characters written to output
precision	Yes	Depending on the conversion used, it will restrict the number of character in the output string
conversion	No	Indicates how the argument will be formatted

The following table shows the general meaning of the conversion field character:

Conversion Character	Description
'b', 'B'	Boolean
'h', 'H'	Hexadecimal
's', 'S'	String
'c', 'C'	Character
'd'	Decimal integer
'o'	Octal integer
'x', 'X'	Hexadecimal integer
'e', 'E'	Scientific notation
'f'	Decimal number
'g', 'G'	Scientific notation or decimal format depending on the precision and the value after rounding.
'a', 'A'	Hexadecimal floating-point number with a significand and an exponent
't', 'T'	Date and time conversion characters
'%'	Literal '%' ('\u0025')
'n'	Platform-specific line separator

The following program illustrates the usage of the printf method:

```
public class PrintfDemo {  
  
    public static void main(String args[]) {  
        int num1 = 562;  
        int num2 = 13;  
        double result;  
        String text = " divided by ";  
  
        result = num1/ (double) num2;  
  
        // Using the argument index field  
        System.out.printf("%1$d divided by %3$d is %2$f\n",  
                           num1, result, num2);  
  
        // Not using the argument index field  
        System.out.printf("%d divided by %d is %f\n",num1, num2, result);  
  
        // Using the %n specifier  
        System.out.printf("%d divided by %d is %5.2f%n",num1, num2, result);  
  
        // Using a small width specifier  
        System.out.printf("%d divided by %d is %1.2f%n",num1, num2, result);  
    }  
}
```

```
// Using a %s field
System.out.printf("%d%s%d is %1.2f%n",num1, text, num2, result);

// Using %o, %d, and %x specifiers with a single argument
System.out.printf("Base 8: %1$o Base 10: %1$d base 16: %1$x%n",
                  num1);
    }
}
```

Input Using the Scanner Class

The Scanner class provides an easier technique for reading data in from an input stream. The constructors use an input stream as its argument. It works for all streams that implement the Readable interface. Standard input can be used as follows:

```
Scanner sc = new Scanner(System.in);
```

In the following example, the file, data.txt, is used as the input source:

```
Scanner sc = new Scanner(new File("data.txt"));
```

Once an instance of the Scanner class has been created there are a number of methods that can be used to read in information. Most of these are of the form:

```
nextDataType()
```

where the data types correspond to Java's primitive data types. A series of *hasNextDataType* methods are used to determine if there exist data of the specified type in the input stream.

To read in a series of integers from a file, the following code sequence can be used:

```
public static void main(String args[]) throws IOException {
    Scanner sc = new Scanner(new File("data.txt"));
    int i;

    while(sc.hasNextInt()) {
        i = sc.nextInt();
        System.out.println(i);
    }
}
```

The hasNext and next methods act on string data.

```
sc = new Scanner(new File("data.txt"));
String s;
while(sc.hasNext()) {
    s = sc.next();
    System.out.println(s);
}
```

One line at a time can be read using the Scanner class using the nextLine method. The CR/LF is not returned by the nextLine method.

```
sc = new Scanner(new File("data.txt"));
while(sc.hasNext()) {
    s = sc.nextLine();
    System.out.println(s);
}
```

It is also possible to read the entire input stream with a single statement. The read includes CR/LF.

```
s = new Scanner(new File("data.txt")).useDelimiter("\\A").next();
System.out.println(s);
```

BufferedReader

BufferedReader is used to buffer character streams. The input is buffered for efficiency reasons. The constructor for this class uses an instance of InputStreamReader as an argument.

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
```

Or perhaps more succinctly:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

Methods of BufferedReader include:

- **read** – Returns a single character
- **readLine** – Returns a line (null if end of file)

```
import java.io.*;

public class ConsoleInput {
    public static void main(String args[]) throws IOException {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.println("Enter your name: ");
        String name = br.readLine();
        System.out.println("Hi " + name);
    }
}
```

Reading an Integer

Reading an integer from standard input is not a straightforward task in Java. It involves reading a line of text, extracting the integer characters, and then converting those characters into an int data type value. Along the way, exceptions need to be catch and handled. One approach for doing this is illustrated below.

```
public static int readInteger() {
    String line;

    line = readString();
    return Integer.parseInt(line);
}

public static String readString() {
    InputStreamReader isr;
    BufferedReader br;
    String line;

    try {
        isr = new InputStreamReader(System.in);
        br = new BufferedReader(isr);
        line = br.readLine();
        return line;
    }
    catch (IOException e) {}
    return "Error";
}
```

BufferedWriter

BufferedWriter is used to buffer character output streams. The constructor for this class uses an instance of Writer as an argument.

```
BufferedWriter br = new BufferedWriter(instance of Writer);
```

Methods of BufferedWriter include:

- **write(*String*)** – Writes out a string
- **newLine** – Writes out a CR/LF
- **flush** – Flushes the output buffer

File Streams

The `FileReader` and `FileWriter` classes provide a means to read and write to files. They normally use either a `String` or a `File` object for their constructors. The `File` class is a system independent way of representing a file. The constructor can throw a `FileNotFoundException` that must be caught.

```
FileReader fr = new FileReader("C:\\system.log");  
BufferedReader br = new BufferedReader(fr);
```

```
FileWriter fw = new FileWriter("C:\\system.bak");  
BufferedWriter bw = new BufferedWriter(fw);
```

```
String line;  
line = br.readLine();
```

```
while (line != null) {  
    bw.write(line);  
    bw.newLine();  
    line = br.readLine();  
}
```

StreamTokenizer

The `StreamTokenizer` class is used to parse an input stream into a series of tokens. A token is a series of characters that are separated by delimiters. A delimiter can be any set of characters. They are frequently white spaces (blanks, tabs, CR/LF, etc.). The constructor for this class uses a character input stream that may be buffered.

```
FileReader fr = new FileReader("C:\\system.log");  
BufferedReader br = new BufferedReader(fr);  
StreamTokenizer t = new StreamTokenizer(br);
```

Methods are then applied to the stream to parse the tokens. **`nextToken()`** – Gets the type of the next token. **`lineno()`** – Returns the current line number. The token object types are:

- **`TT_EOF`** – End of file
- **`TT_EOL`** – End of line
- **`TT_NUMBER`** – A number token
- **`TT_WORD`** – A word token

StreamTokenizer Example

The StreamTokenizer instance has a number of methods that assist in parsing the character stream.

- **eolIsSignificant()** – Determines whether the end of line should be treated specially
- **whitespaceChars()** – Determines the characters that make up white spaces
- **commentChar()** – Specifies which characters will be treated as comments

In addition, the StreamTokenizer has two properties that assist in determining the type of token:

- **nval** – The numeric value of the token if it is a number
- **sval** – The string value of the token if it is a word

In this example the file is parsed.

```
FileReader fr = new FileReader("C:\\system.log");
BufferedReader br = new BufferedReader(fr);
StreamTokenizer t = new StreamTokenizer(br);

int tokenType;
while (true) {
    tokenType = t.nextToken();
    switch (tokenType) {
        case TT_EOF: break;
        case TT_NUMBER: System.out.println(t.nval);
        case TT_WORD: System.out.println(t.sval);
    }
}
```

Section 2: File System Access

File Class

The File class provides a means to access and manipulate the underlying file system. Methods exist that obtain information about the file system and how to modify it. An instance of File can represent either a file or a directory.

```
public File(String path);
public File(String path, String name);
public File(File dir, String name);
```

Commonly used methods include:

- **canRead** – Indicates whether the file can be read or not
- **canWrite** - A boolean method that indicates whether the file can be written to
- **delete** – Used to delete a file
- **exists** – Determines whether a file or directory exists
- **getPath** – Returns the path to the file or directory as a String
- **isDirectory** and **isFile**– A boolean method that determines if the object is a directory or file
- **length** – Returns the length of a file as a long number
- **list** – A method that returns an array of Strings representing the files in a directory
- **mkdir** – Used to create a directory
- **renameTo** – Renames a file or directory

```
import java.io.*;

public class FileDemo {
    public static void main(String args[]) {
        File file = new File(args[0]);
        if (file.exists()) {
            file.delete();
            System.out.println("File: " + args[0] + " deleted!");
        } else {
            System.out.println("File: " + args[0] + " does not exist!");
        }
    }
}
```

List Method

The list method returns an array of Strings containing the filenames for a specific directory. It may be used with an argument that implements the FilenameFilter interface. This provides a means of creating an arbitrarily complex set of criteria for selecting files.

```
import java.io.*;
public class FilenameFilter1 {
    public static void main(String args[]) {
        String fileNameList[];

        File file = new File(".");
        fileNameList = file.list();
        for (int i=0; i<fileNameList.length; i++) {
            System.out.println(fileNameList[i]);
        }
    }
}
```

FilenameFilter

The `FilenameFilter` is an interface that is used to filter the list of files returned by the **list** method based on a specific set of criteria. The `FilenameFilter` interface has a single abstract method.

```
public abstract interface FilenameFilter {  
    public abstract boolean accept(File dir, String name);  
}
```

When the **list** method is invoked with an object that implements this interface, the **accept** method is invoked once for each file found within the selected directory. The **accept** method will then return either true or false. If it returns true, the file name is returned by the list method. If a false is returned by the **accept** method, the file name is not returned by the **list** method.

```
import java.io.*;  
  
public class FilenameFilter2 {  
    public static void main(String args[]) {  
        String fileNameList[];  
        FilenameFilter javaFilter = new FileFilter();  
  
        File file = new File(".");  
        fileNameList = file.list(javaFilter);  
        for (int i=0; i<fileNameList.length; i++) {  
            System.out.println(fileNameList[i]);  
        }  
    }  
}  
  
class FileFilter implements FilenameFilter {  
    public boolean accept(File dir, String name) {  
        if ((name.endsWith(".java")) || (name.endsWith(".class"))) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

Formatting Output

Formatting of numeric output is done using the Format Class. Java provides formatting capabilities through its abstract Format class that is part of the java.text package. The NumberFormat Class is derived from Format . The DecimalFormat Class is derived from NumberFormat and is normally used for long and double numbers. A **getInstance** type of method is used with NumberFormat.

```
public static NumberFormat getInstance();  
public static NumberFormat getNumberInstance();  
public static NumberFormat getPercentInstance();
```

Several methods exist to format numbers

```
public void setMaximumFractionDigits(int newValue);  
public void setMaximumIntegerDigits(int newValue);  
public void setMinimumFractionDigits(int newValue);  
public void setMinimumIntegerDigits(int newValue);
```

The **format** method is used to convert a long or double number to a String. The **parse** method converts a String to a Number object.

```
public final String format(double number);  
public final String format(long number);  
public Number parse(String text) throws ParseException;
```

NumberFormatDemo

```
import java.text.*;

public class NumberFormatDemo {
    public static void main(String args[]) {
        NumberFormat nf = NumberFormat.getInstance();
        System.out.println(nf.format(12));
        System.out.println(nf.format(1234567));
        System.out.println(nf.format(12.34567));
        System.out.println(nf.format(123456.7));

        nf.setMaximumFractionDigits(2);
        nf.setMaximumIntegerDigits(5);

        System.out.println();
        System.out.println(nf.format(12));
        System.out.println(nf.format(1234567));
        System.out.println(nf.format(12.34567));
        System.out.println(nf.format(123456.7));
    }
}
```

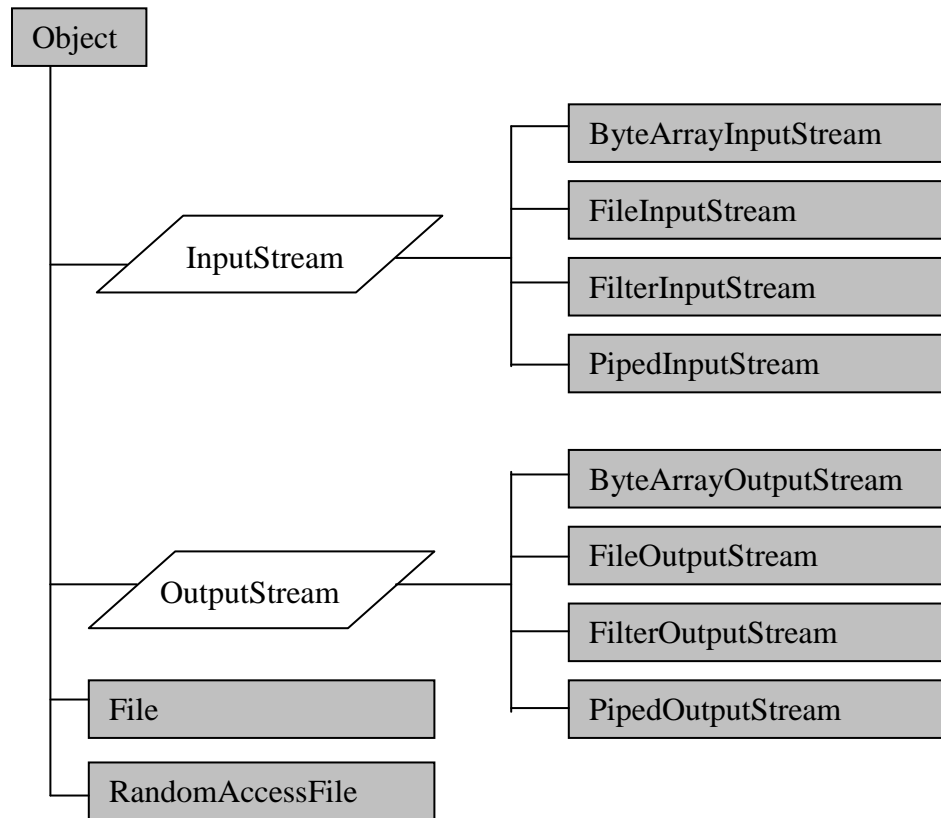
```
12
1,234,567
12.346
123,456.7
```

```
12
34,567
12.35
23,456.7
```

Section 3: InputStream and OutputStream

Input and Output

The `InputStream` and `OutputStream` classes have been around since the start of Java. Since then numerous other classes have been added to improve the I/O capabilities of Java. The `InputStream` and `OutputStream` classes are still useful.



InputStream and OutputStream

Derived InputStream classes include:

- ByteArrayInputStream – Used to read from a byte array in memory
- FileInputStream – Allows a file to be opened for input
- FilterInputStream – The basis for buffered input
- PipedInputStream – Used to communicate between threads

Derived OutputStream classes include:

- ByteArrayOutputStream – Used to write to an array of bytes in memory
- FileOutputStream – Permits writing to a file
- FilterOutputStream – The foundation for buffered output
- PipedOutputStream – Used to communicate between threads

InputStream Methods

Important methods of the InputStream class:

- **read** – Reads bytes from the input stream

```
public abstract int read() throws IOException  
public int read(byte[] b) throws IOException  
public int read(byte[] b, int off, int length) throws IOException
```

Each of these methods return a `-1` if EOF has been reached. The first method returns a single byte. The latter two methods return the number of bytes read in the byte array argument. The third method allows for specification of the starting position into the array and the maximum number of bytes to read into the array. With the latter two methods, if the array is filled, the **read** method returns.

- **available** - Returns the number of bytes left in the input stream

```
public int available() throws IOException;
```

- **markSupported** - indicates whether the current position can be marked
- **mark** - will mark the position
- **reset** - will reposition the file to the previously marked position

The *readlimit* argument specifies how far ahead (bytes) it is possible to read before the mark becomes invalidated.

```
public synchronized void mark(int readlimit);  
public boolean markSupported ();  
public synchronized void reset() throws IOException
```

- **skip** - Used to skip over input bytes as specified by the long argument

```
public long skip(long n) throws IOException;
```

- **close** - closes the input stream

```
public void close() throws IOException;
```

InputStream Example

The `readString` method will read a line from standard input, that is, the keyboard. In the code that follows, `System.in` is a predefined object that corresponds to standard input. An `InputStreamReader` object is created which will take the bytes from the keyboard and convert them into characters. This object is used as the argument for the `BufferedReader` constructor. This object performs buffered input which make it more efficient and also provides a **`readLine`** method that reads in the entire line up to but not including the carriage return.

The `readInteger` method will use the string returned from the `readString` method and convert it to an integer, which it returns. The static **`parseInt`** method of the `Integer` class converts the string to an integer.

```
public static int readInteger() {
    String line;

    line = Calculate.readString();
    return Integer.parseInt(line);
}

public static String readString() {
    InputStreamReader isr;
    BufferedReader br;
    String line;

    try {
        isr = new InputStreamReader(System.in);
        br = new BufferedReader(isr);
        line = br.readLine();
        return line;
    }
    catch (IOException e) {}
    return "Error";
}
```

OutputStream Class

Important methods of the OutputStream class include:

- **write** – Used to write a stream of bytes

```
public abstract void write(byte b) throws IOException;  
public void write(byte[] b) throws IOException;  
public void write(byte[] b, int offset, int length) throws IOException;
```

- **close** - Used to close the output stream
- **flush** – Used to forces any buffered output to be written

Whenever an input stream is opened it should be closed as soon as it is no longer needed. The **close** method performs this operation. When an application terminates, all open streams are automatically closed. However, keeping a stream open when it is not needed is not a good programming practice and may consume system resources.

Buffered I/O creates a buffer where data is stored temporarily before it is shipped to application. The buffering process will generally improve the performance of I/O. However, the buffer is not sent to the application in the case of buffered input or out of the program with buffered output unless the buffer is full. To force the transmission of a partially filled buffer, the **flush** method can be used.

Byte Array Streams

The `ByteArrayInputStream` and the `ByteArrayOutputStream` classes are used to read and write from a byte array in memory. Keeping data in memory can improve the overall I/O performance. The idea is that memory is used as a temporary location where data is read from or written to. Reading and writing from memory is faster than the I/O used against an external device. Once the data is finalized in memory, it is then shipped to the external device in the case of an output stream. Overall the performance of the operation is improved.

```
public ByteArrayInputStream(byte b[] buf);  
public ByteArrayInputStream(byte[] buf, int offset, int length);
```

The constructors for the `ByteArrayOutputStream` create a byte array that is written to. This array will grow as necessary and can be retrieved using the `toByteArray` or `toString` methods.

```
public ByteArrayOutputStream();  
public ByteArrayOutputStream(int size);  
  
public synchronized byte[] toByteArray();  
public String toString();
```

```
import java.io.*;  
  
class ByteArrayOutputStreamDemo {  
    public static void main(String args[]) throws IOException {  
        ByteArrayOutputStream bout = new ByteArrayOutputStream();  
        int i;  
  
        for (i=0; i<50; i++)  
        {  
            bout.write((new String("Line Number: " + i + "\n")).getBytes());  
        }  
        System.out.println(bout.toString());  
    }  
}
```

File Stream Class

File input and output is accomplished using the `FileInputStream` and `FileOutputStream` classes. A constructor is used to open the file. The constructors may use a filename, `File` object or `FileDescriptor` object. File I/O methods are similar to those of the base classes.

The `File` and `File Descriptor` classes will be discussed in more detail later.

```
public FileInputStream(String name) throws FileNotFoundException;  
public FileInputStream(File file) throws FileNotFoundException;  
public FileInputStream(FileDescriptor filedescriptorobject);
```

```
import java.io.*;  
  
class FileStreamDemo {  
    public static void main(String args[]) throws IOException {  
        FileInputStream fin = new FileInputStream(args[0]);  
        FileOutputStream fout = new FileOutputStream(args[1]);  
        byte byteArray[] = new byte[256];  
        int i;  
  
        while ((i = fin.read(byteArray)) != -1) {  
            fout.write(byteArray);  
        }  
    }  
}
```

Piped Streams and Buffered Streams

There are several ways to communicate between threads. The `PipedInputStream` and `PipedOutputStream` classes provide a technique of passing information between threads. An instance of a `PipedInputStream` is normally used as the argument for the corresponding `PipedOutputStream` instance.

The primary purpose of Buffered Streams is to improve I/O performance. The default input buffer size is 2048 bytes. The default size of the buffer is 512 bytes. Data to be read or written is stored in temporary buffers. These buffers are emptied under appropriate conditions. The size of the buffers can be set

Data Streams

Data streams are used for convenient input and output of primitive data types. Data streams are frequently used with Buffered Streams.

```
public DataInputStream(InputStream in);  
public DataOutputStream(OutputStream out);
```

```
FileInputStream fin = new FileInputStream(args[0]);  
BufferedInputStream bin = new BufferedInputStream(fin);  
DataInputStream din = new DataInputStream(bin);
```

The data is written and read in binary format. Methods exist for reading and writing primitive data types.

```
public boolean readBoolean() throws IOException;  
public byte readByte() throws IOException;  
public char readChar() throws IOException;  
public double readDouble() throws IOException;  
public float readFloat() throws IOException;  
public int readInt() throws IOException;  
public String readLine() throws IOException;  
public long readLong() throws IOException;  
public short readShort() throws IOException;
```

Data Stream Example

```
import java.io.*;

public class DataStreamDemo {
    public static void main(String args[]) throws IOException {
        int i;

        FileOutputStream fout = new FileOutputStream(args[0]);
        BufferedOutputStream bout = new BufferedOutputStream(fout);
        DataOutputStream dout = new DataOutputStream(bout);

        for (i=0; i <= 20; i++) {
            dout.writeInt(i);
            dout.writeInt(i*i);
        }
        dout.close();

        FileInputStream fin = new FileInputStream(args[0]);
        BufferedInputStream bin = new BufferedInputStream(fin);
        DataInputStream din = new DataInputStream(bin);

        for (i=0; i <= 20; i++) {
            System.out.println("Number: " + din.readInt() + " Square: " +
                din.readInt());
        }
        din.close();
    }
}
```

PrintStream Class

The `PrintStream` class provides methods to display primitive data types. ASCII output is produced. The `print` and `println` methods are the primary methods used. `System.out` and `System.err` are instances of the `PrintStream` class.

```
public PrintStream(OutputStream out);  
public PrintStream(OutputStream out, boolean autoflush);
```

The `print` and `println` methods are overloaded and will take on any of the primitive data types in addition to the `String` object.

RandomAccessFile Class

Random access is accomplished using instances of the `RandomAccessFile` class. A file can be opened for input and/or output.

```
public RandomAccessFile(String name, String mode) throws IOException;  
public RandomAccessFile(File file, String mode) throws IOException;
```

The mode may be either “r” or “rw”. Important methods include:

- **seek** – This method uses a single long argument to move to that position in the file.
- **getFilePoint** – This method returns a long value indicating the current position within the file.
- **length** – The **length** method returns the length of the file as a long number.
- **readFully** - This method will read the entire contents of the byte array specified as an argument

```
import java.io.*;  
  
public class RandomFileDemo {  
    public static void main(String args[]) throws IOException {  
        int recordNumber;  
        byte buffer[] = new byte[10];  
        int i;  
  
        RandomAccessFile rin = new RandomAccessFile(args[0], "rw");  
  
        for (i=0; i<10; i++) {  
            rin.writeBytes("String " + i);  
        };  
  
        recordNumber = Integer.parseInt(args[1]);  
  
        rin.seek(recordNumber * 10);  
        rin.readFully(buffer);  
        System.out.println(new String(buffer));  
        rin.close();  
    }  
}
```

Section 4: New I/O Classes

New I/O Classes

In version 1.4, Java introduced a number of new I/O classes. These were introduced to provide high-performance, scalable I/O. These classes are found in the java.nio package. Their primary benefit is to be able to handle thousands of open connections with scalability and high performance.

There are four classes of primary interest:

- **Buffer** – Provides for memory-mapped file I/O
- **Charset** – Performs Unicode character conversions
- **Channels** – Is a bi-direction communication pipe
- **Selectors** – Allows multiplex asynchronous I/O operations for a thread

In this section we will only discuss the Buffer class and its derivatives.

Buffer Class

The Buffer class is an abstract base class for a number of specialized classes. They all represent a linear, finite sequence of some primitive data type. They were introduced to avoid some of the performance overhead associated with garbage collection.

Derived Buffer classes include:

- ByteBuffer
- MappedByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

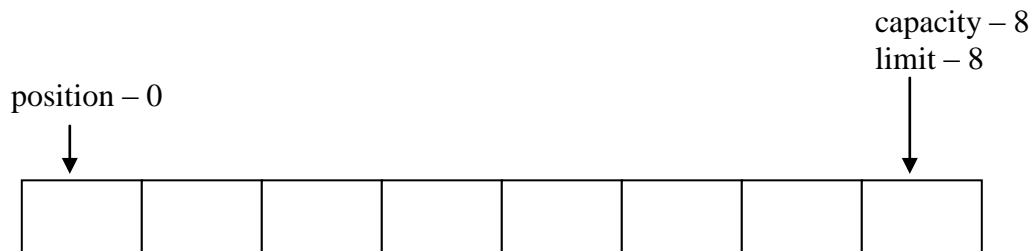
Buffer Characteristics

When a buffer is created, its size is fixed and cannot be changed. There are three terms used to describe a buffer:

- **Capacity** - The *capacity* is the number of elements that the buffer can hold. The size of the buffer at creation determines its capacity.
- **Limit** - A buffer's *limit* is an index into the buffer. It is the first element that should not be read or written. The limit will never exceed its capacity.
- **Position** - A buffer's *position* is the index of the next element to be read or written. This value is never negative and will never exceed its limit.

When a buffer is created, the limit and capacity are the same and the position is set to 0. The static `allocate` method is used to create a buffer. The `ByteBuffer` class has an `allocateDirect` method.

```
ByteBuffer buffer = ByteBuffer.allocateDirect(8);
```



The following invariant holds for the position, limit, and capacity values:

$$0 \leq \text{position} \leq \text{limit} \leq \text{capacity}$$

A buffer may be either direct or non-direct. A direct buffer allocates contiguous storage and uses native code to access the data. This is the preferred buffer type. A non-direct buffer uses Java array accessors which can be slower.

However, the allocation and deallocation costs of a direct buffer may be higher than that of a non-direct buffer. Sun recommends that direct buffers be used for large, long term needs.

View Buffer

The only class that supports the **allocateDirect** method in JDK 1.4 is the `ByteBuffer` class. However this does not mean that other buffer classes can't be direct. A *view* buffer is a buffer whose content is backed by the byte buffer.

Changes to either the byte buffer or the view buffer will affect the other. The position, limit and marks are independent of each other. A view buffer is created using a method such the **asIntBuffer** method. This method creates an instance of an `IntBuffer` that is backed by a `ByteBuffer`.

```
ByteBuffer buffer = ByteBuffer.allocateDirect(size*4);
IntBuffer ib = buffer.asIntBuffer();
```

Notice that the argument for the **allocateDirect** method multiplies `size`, the number of integers need, by 4 (the number of bytes used to store an integer). This means that `ib` can hold up to `size` integers.

asPrimitiveDatatypeBuffer methods exist for all of the primitive types except for `boolean`.

The view buffer is not indexed in terms of bytes but in terms of the new buffer type. There are bulk get and put methods based on the primitive data type of the view buffer. The view buffer is direct if the backing buffer is direct.

The **isDirect** method can be used to determine if the view buffer is direct or not.

put and get Methods

put and **get** methods are used to access a buffer. These methods are either absolute or relative. An absolute method specifies the position in the buffer to access. A relative method uses the next position.

- `get()` – Relative get
- `get(int index)` – Absolute get
- `put(long value)` – Relative put (For the `LongBuffer` class)
- `put(int index, long value)` – Absolute put (For the `LongBuffer` class)

Once a buffer contains data it needs to be prepared. This is done using the **flip** method. This method will set limit to the current position and set the position to 0. The buffer can now be read.

```
IntBuffer iBuffer = IntBuffer.allocate(8);

iBuffer.put(5);
iBuffer.put(6);
iBuffer.put(7);

iBuffer.flip();

System.out.println(iBuffer.get());
System.out.println(iBuffer.get());
System.out.println(iBuffer.get());
```

If a get operation tries to read past the limit a `BufferUnderFlow` exception is generated. If a put operation tries to write past the limit a `BufferOverflowException` is generated. An absolute operation that exceeds the limit will generate an `IndexOutOfBoundsException`.

The **clear** method will set the limit to the capacity and the position to 0. It is now ready for new operations.

The **rewind** method makes the buffer available for rereading. The limit is left alone and position is set to 0.

Buffers are not thread safe. To make it thread safe, access to the buffer must be controlled through synchronization.

Memory-Mapped Files

The `MappedByteBuffer` class is a way to map part or all of a file directly into memory. This will make access to the file faster than traditional IO techniques in Java. As of Java 1.4, a `FileChannel` is needed to create a `MappedByteBuffer` object. A `FileChannel` is created from a `FileInputStream` or a `FileOutputStream` object.

In the example, a `MappedByteBuffer` is mapped to `data.txt` and is used to count the number of 'a's in the file. The `map` method of `FileChannel` creates the buffer. The file can be opened for read, write or read/write operations.

```
try {
    File f = new File("data.txt");
    FileInputStream fis = new FileInputStream(f);
    FileChannel fc = fis.getChannel();

    int fileSize = (int) fc.size();
    MappedByteBuffer buf =
        fc.map(FileChannel.MapMode.READ_ONLY, 0, fileSize);

    for(i=0; i<fileSize; i++) {
        if((char)buf.get(i) == 'a') {
            aCount++;
        }
    }

    System.out.print("A Count: " + aCount);
}

catch (IOException e) {}
```

Summary

- The File Class is used to obtain information about the underlying file system
- Formatting of numeric data is performed using the NumberFormat Class
- The Reader and Writer classes provide a new and better implementation for character I/O
- The InputStream and OutputStream classes form the basis for most of the commonly used input and output operations within Java
- The RandomAccessFile Class provides random access capability