Using I/O

Java's I/O is Built upon Streams

- Java programs perform I/O through streams.
- A stream is an abstraction that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- This means that an input stream can abstract many kinds of input: from a disk file, a keyboard, or a network socket.
- Likewise, an output stream may refer to the console, a disk file, or a network connection.
- Java implements streams within class hierarchies defined in the java.io package.

Byte Streams and Character Streams

- Java defines two types of streams: byte and character.
- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.
- Character streams provide a convenient means for handling input and output of characters.
- They use Unicode and, therefore, can be internationalized.
- Also, in some cases, character streams are more efficient than byte streams.

The Byte Stream Classes

- Byte streams are defined by using two class hierarchies.
- At the top are two abstract classes: InputStream and OutputStream.
- Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers.
- The byte stream classes in java.io are shown in the next table.
- The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement.
- Two of the most important are read( ) and write( ), which, respectively, read and write bytes of data.
- Each has a form that is abstract and must be overridden by derived stream classes.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

The Character Stream Classes

- Character streams are defined by using two class hierarchies.
- At the top are two abstract classes: Reader and Writer.

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

The Predefined Streams

- All Java programs automatically import the java.lang package.
- This package defines a class called System, which encapsulates several aspects of the run-time environment.
- System also contains three predefined stream variables: in, out, and err.
- These fields are declared as public, static, and final within System.
- This means that they can be used by any other part of your program and without reference to a specific System object.

Using the Byte Streams

Let's explore FileInputStream and FileOutputStream by examining an example program named CopyBytes, which uses byte streams to copy xanadu.txt, one byte at a time.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
	public static void main(String[] args) throws IOException {

		FileInputStream in = null;
		FileOutputStream out = null;

		try {
			in = new FileInputStream("xanadu.txt");
			out = new FileOutputStream("outagain.txt");
			int c;

			while ((c = in.read()) != -1) {
				out.write(c);
			}
		} finally {
			if (in != null) {
				in.close();
			}
			if (out != null) {
				out.close();
			}
		}
	}
}
```

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time.

Reading and Writing Files

Two of the most often-used stream classes are FileInputStream and FileOutputStream, which create byte streams linked to files.

```
import java.io.*;

class ShowFile {
	public static void main(String args[]) {
		int i;
		FileInputStream fin;
		// First, confirm that a filename has been specified.
		if (args.length != 1) {
			System.out.println("Usage: ShowFile filename");
			return;
		}
		// Attempt to open the file.
		try {
			fin = new FileInputStream(args[0]);
		} catch (FileNotFoundException e) {
			System.out.println("Cannot Open File");
			return;
		}
		// At this point, the file is open and can be read.
		// The following reads characters until EOF is encountered.
		try {
			do {
				i = fin.read();
				if (i != -1)
					System.out.print((char) i);
			} while (i != -1);
		} catch (IOException e) {
			System.out.println("Error Reading File");
		}
		// Close the file.
		try {
			fin.close();
		} catch (IOException e) {
			System.out.println("Error Closing File");
		}
	}
}
```

Automatically Closing a File

- JDK 7 added a new feature that offers another way to manage resources, such as file streams, by automating the closing process.
- This feature, sometimes referred to as automatic resource management, or ARM for short, is based on an expanded version of the try statement.
- The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed.

```
try (resource-specification) {
        // use the resource
}
```

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class CopyBytes {
	public static void main(String[] args) {
		// The following code uses a try-with-resources statement to open
		// a file and then automatically close it when the try block is
left.
		try (FileInputStream fin = new FileInputStream(args[0])) {
			do {
				i = fin.read();
				if (i != -1)
					System.out.print((char) i);
			} while (i != -1);
		} catch (FileNotFoundException e) {
			System.out.println("File Not Found.");
		} catch (IOException e) {
			System.out.println("An I/O Error Occurred");
		}
	}
}
```

Random-Access Files

Random access files permit nonsequential, or random, access to a file's contents. To access a file randomly, you open the file, seek a particular location, and read from or write to that file.

This functionality is possible with the SeekableByteChannel interface. The SeekableByteChannel interface extends channel I/O with the notion of a current position. Methods enable you to set or query the position, and you can then read the data from, or write the data to, that location. The API consists of a few, easy to use, methods:

- position – Returns the channel's current position
- position(long) – Sets the channel's position
- read(ByteBuffer) – Reads bytes into the buffer from the channel
- write(ByteBuffer) – Writes bytes from the buffer to the channel
- truncate(long) – Truncates the file (or other entity) connected to the channel

The following code snippet opens a file for both reading and writing by using one of the newByteChannel methods. The SeekableByteChannel that is returned is cast to a FileChannel. Then, 12 bytes are read from the beginning of the file, and the string "I was here!" is written at that location. The current position in the file is moved to the end, and the 12 bytes from the beginning are appended. Finally, the string, "I was here!" is appended, and the channel on the file is closed.

```
String s = "I was here!\n";
byte data[] = s.getBytes();
ByteBuffer out = ByteBuffer.wrap(data);

ByteBuffer copy = ByteBuffer.allocate(12);

try (FileChannel fc = (FileChannel.open(file, READ, WRITE))) {
      // Read the first 12
      // bytes of the file.
      int nread;
      do {
            nread = fc.read(copy);
      } while (nread != -1 && copy.hasRemaining());

      // Write "I was here!" at the beginning of the file.
      fc.position(0);
      while (out.hasRemaining())
            fc.write(out);
      out.rewind();

      // Move to the end of the file. Copy the first 12 bytes to
      // the end of the file. Then write "I was here!" again.
      long length = fc.size();
      fc.position(length - 1);
      copy.flip();
      while (copy.hasRemaining())
            fc.write(copy);
      while (out.hasRemaining())
            fc.write(out);
} catch (IOException x) {
      System.out.println("I/O Exception: " + x);
}
```

Using Java's Character-Based Streams

The following Java program reads data from a particular file using FileReader and writes it to another, using FileWriter.

```
// Creating FileReader object
File file = new File("D:/myFile.txt");
FileReader reader;
try {
	reader = new FileReader(file);
	char chars[] = new char[(int) file.length()];
	// Reading data from the file
	reader.read(chars);
	// Writing data to another file
	File out = new File("D:/CopyOfmyFile.txt");
	FileWriter writer = new FileWriter(out);
	// Writing data to the file
	writer.write(chars);
	writer.flush();
} catch (FileNotFoundException e) {
	// TODO Auto-generated catch block
	e.printStackTrace();
} catch (IOException e) {
	// TODO Auto-generated catch block
	e.printStackTrace();
} finally {
	System.out.println("Data successfully written in the specified file");
}
```

Using Java's Type Wrappers to Convert Numeric to Strings

Sometimes, you need to convert a number to a string because you need to operate on the value in its string form. All classes inherit a method called toString from the Object class. The type-wrapper classes override this method to provide a reasonable string representation of the value held by the number object.

The following program, ToStringDemo (in a .java source file), uses the toString method to convert a number to a string. Next, the program uses some string methods to compute the number of digits before and after the decimal point:

```
double d = 858.48;
String s = Double.toString(d);

int dot = s.indexOf('.');
System.out.println(s.substring(0, dot).length() + " digits before decimal
point.");
System.out.println(s.substring(dot + 1).length() + " digits after decimal
point.");
```

The toString method called by this program is the class method. Each of the number classes has an instance method called toString, which you call on an instance of that type.

You don't have to explicitly call the toString method to display numbers with the System.out.println method or when concatenating numeric values to a string. The Java platform handles the conversion by calling toString implicitly.