

Introduction to Java™



Module 10: Stream I/O and Files

Introduction to Java IO

- The Java library designers attacked the problem by creating **lots of classes**.
- In fact, there are **so many classes** for Java's IO system that it can be **intimidating at first**.
- There has also been **significant changes** in the IO library **between Java 1.0, Java 1.1 and Java 2**.
 - Instead of simply replacing the old library with a new one, the designers at Sun **extended** the old library and added the new one alongside it.
 - As a result you can sometimes end up **mixing the old and new libraries** and creating even more intimidating code.
 - The last extension to the I/O library made in Java 2 (since 1.4) follows a different approach.
 - ✓ **New library** named new I/O (**java.nio** package)

Input and Output



- The Java library classes for IO are divided by input and output.
- By inheritance, all classes derived from **InputStream** or **Reader** have basic methods called **read()** for reading a single byte or array of bytes.
- Likewise, all classes derived from **OutputStream** or **Writer** have basic methods called **write()** for writing a single byte or array of bytes.

Input and Output

- However, you won't generally use these methods;
they exist so more sophisticated classes can use them as they provide a more useful interface.
- Thus, you'll rarely create your stream object by using a single class, but instead will layer multiple objects together to provide your desired functionality.
- The fact that you create more than one object to create a single resulting stream is the primary reason that Java's stream library is confusing.

Types of InputStream



- **InputStream**'s job is to represent classes that produce input from different sources.
- These sources can be:
 - An **array of bytes**
 - A **String** object
 - A **file**
 - A "**pipe**" which works like a physical pipe: you put things in one end and they come out the other
 - A **sequence of other streams**, so you can collect them together into a single stream
 - Other sources, such as an Internet connection

Table 10-1: Types of InputStream

Class	Function	Constructor Arguments
		How to use it
ByteArrayInputStream	Allows a buffer in memory to be used as an InputStream .	The buffer from which to extract the bytes.
		As a source of data. Connect it to a FilterInputStream object to provide a useful interface.
StringBufferInputStream	Converts a String into an InputStream .	A String . The underlying implementation actually uses a StringBuffer .
		As a source of data. Connect it to a FilterInputStream object to provide a useful interface.
File-InputStream	For reading information from a file.	A String representing the file name, or a File or FileDescriptor object.
		As a source of data. Connect it to a FilterInputStream object to provide a useful interface.
PipedInputStream	Produces the data that's being written to the associated PipedOutput-Stream . Implements the "piping" concept.	PipedOutputStream
		As a source of data in multithreading. Connect it to a FilterInputStream object to provide a useful interface.
SequenceInputStream	Covers two or more InputStream objects into a single InputStream .	Two InputStream objects or an Enumeration for a container of InputStream objects.
		As a source of data. Connect it to a FilterInputStream object to provide a useful interface.
FilterInputStream	Abstract class which is an interface for decorators that provide useful functionality to the other InputStream classes. See Table 10-3.	See Table 10-3.
		See Table 10-3.

Table 10-2: Types of OutputStream

Class	Function	Constructor Arguments
		How to use it
ByteArray-OutputStream	Creates a buffer in memory. All the data that you send to the stream is placed in this buffer.	Optional initial size of the buffer.
		To designate the destination of your data. Connect it to a FilterOutputStream object to provide a useful interface.
File-OutputStream	For sending information to a file.	A String representing the file name, or a File or FileDescriptor object.
		To designate the destination of your data. Connect it to a FilterOutputStream object to provide a useful interface.
Piped-OutputStream	Any information you write to this automatically ends up as input for the associated PipedInputStream . Implements the “piping” concept.	PipedInputStream
		To designate the destination of your data for multithreading. Connect it to a FilterOutputStream object to provide a useful interface.
Filter-OutputStream	Abstract class which is an interface for decorators that provide useful functionality to the other OutputStream classes. See Table 10-4.	See Table 10-4.
		See Table 10-4.

Table 10-3. Types of FilterInputStream

Class	Function	Constructor Arguments
		How to use it
Data-InputStream	Used in concert with DataOutputStream , so you can read primitives (int, char, long, etc.) from a stream in a portable fashion.	InputStream
		Contains a full interface to allow you to read primitive types.
Buffered-InputStream	Use this to prevent a physical read every time you want more data. You're saying "Use a buffer."	InputStream , with optional buffer size.
		This doesn't provide an interface <i>per se</i> , just a requirement that a buffer be used. Attach an interface object.
LineNumber-InputStream	Keeps track of line numbers in the input stream; you can call getLineNumber() and setLineNumber(int) .	InputStream
		This just adds line numbering, so you'll probably attach an interface object.
Pushback-InputStream	Has a one byte push-back buffer so that you can push back the last character read.	InputStream
		Generally used in the scanner for a compiler and probably included because the Java compiler needed it. You probably won't use this.

Reading From an InputStream With FilterInputStream

- The **FilterInputStream** classes accomplish two significantly different things.
- **DataInputStream** allows you to read different types of primitive data as well as **String** objects.
 - All the methods start with "read," such as **readByte()**, **readFloat()**, etc.
- The remaining classes modify the way an **InputStream** behaves internally:
 - whether it's **buffered or unbuffered**,
 - if it **keeps track of the lines** it's reading,
 - and whether you **can push back** a single character.
- The last two classes look a lot like support for building a compiler, so you probably won't use them in general programming.
- You'll probably **need to buffer** your input almost every time

Table 10-4. Types of FilterOutputStream

Class	Function	Constructor Arguments
		How to use it
Data-OutputStream	Used in concert with DataInputStream so you can write primitives (int, char, long, etc.) to a stream in a portable fashion.	OutputStream
		Contains full interface to allow you to write primitive types.
PrintStream	For producing formatted output. While DataOutputStream handles the <i>storage</i> of data, PrintStream handles <i>display</i> .	OutputStream , with optional boolean indicating that the buffer is flushed with every newline.
		Should be the "final" wrapping for your OutputStream object. You'll probably use this a lot.
Buffered-OutputStream	Use this to prevent a physical write every time you send a piece of data. You're saying "Use a buffer." You can call flush() to flush the buffer.	OutputStream , with optional buffer size.
		This doesn't provide an interface <i>per se</i> , just a requirement that a buffer is used. Attach an interface object.

Writing to an OutputStream With FilterOutputStream

- All the methods start with "write," such as `writeByte()`, `writeFloat()`, etc.
- If you want to do true formatted output, for example, to the console, use a `PrintStream`.
- The `System.out` static object is a `PrintStream`.
- The two important methods in `PrintStream` are `print()` and `println()`, which are overloaded to print out all the various types.
- The difference between `print()` and `println()` is that the latter adds a newline when it's done.
- `BufferedOutputStream` is a modifier and tells the stream to use buffering so you don't get a physical write every time you write to the stream.
 - You'll probably **always want to use this** with files, and possibly console IO.

Off by Itself: RandomAccessFile



- o **RandomAccessFile** is used for files containing records of known size so that you can move from one record to another using **seek()**, then read or change the records.
- o The records don't have to be the same size; you just have to be able to determine how big they are and where they are placed in the file.
- o **RandomAccessFile** is not part of the **InputStream** or **OutputStream** hierarchy.
 - it's a completely separate class, written from scratch, with all of its own (mostly native) methods.

Off by Itself: RandomAccessFile

- The reason for this may be that **RandomAccessFile** has essentially different behaviour than the other IO types, since you can move forward and backward within a file.
- It stands alone, as a direct descendant of **Object**.
- Essentially, a **RandomAccessFile** works like a **DataInputStream** pasted together with a **DataOutputStream** and the methods
 - **getFilePointer()** to find out where you are in the file,
 - **seek()** to move to a new point in the file,
 - and **length()** to determine the maximum size of the file.

Off by Itself: RandomAccessFile



- o In addition, the constructors require a **second argument** (identical to `fopen()` in C) indicating whether you are just randomly reading ("**r**") or reading and writing ("**rw**").
- o There's **no support** for write-only files.

The File class



- o The **File** class has a deceiving name - you might think it refers to a file, but it doesn't.
- o It can represent either the *name* of a particular file or the *names* of a set of files in a directory.
- o "FilePath" would have been a better name.

A Directory Lister



- Suppose you'd like to see a **directory listing**.
- The **File** object can be listed in two ways.
 - If you call **list()** with no arguments, you'll get the full list that the **File** object contains.
 - However, if you want a restricted list, for example, all of the files with an extension of **.java**, then you use a "**directory filter**," which is a class that tells how to select the **File** objects for display.

Example



```
//: DirList.java
// Displays directory listing
import java.io.*;
public class DirList {
    public static void main(String[] args) {
        try {
            File path = new File(".");
            String[] list;
            if(args.length == 0)
                list = path.list();
            else
                list = path.list(new DirFilter(args[0]));
            for(int i = 0; i < list.length; i++)
                System.out.println(list[i]);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Example



```
class DirFilter implements FilenameFilter {
    String afn;
    DirFilter(String afn) { this.afn = afn; }
    public boolean accept(File dir, String name) {
        // Strip path information:
        String f = new File(name).getName();
        return f.indexOf(afn) != -1;
    }
}
```

Example



- Keep in mind that this is a **simple string search** and does **not have regular expression** "wildcard" matching such as "fo?.b?r*" which is much more difficult to implement.
- The **list()** method returns an array. You can query this array for its length and then move through it selecting the array elements.
- This ability to easily pass an array in and out of a method is a tremendous improvement over the behaviour of C and C++.

Checking for and Creating Directories



- o The **File** class is more than just a representation for an existing directory path, file, or group of files.
- o You can also use a **File** object to create a **new directory** or an entire directory path if it doesn't exist.
- o You can also look at the **characteristics of files** (size, last modification date, read/write), see whether a **File** object represents a file or a directory, and delete a file.

Example



```
//: MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
import java.io.*;

public class MakeDirectories {
    private final static String usage =
        "Usage:MakeDirectories path1 ...\n" +
        "Creates each path\n" +
        "Usage:MakeDirectories -d path1 ...\n" +
        "Deletes each path\n" +
        "Usage:MakeDirectories -r path1 path2\n" +
        "Renames from path1 to path2\n";
    private static void usage() {
        System.err.println(usage);
        System.exit(1);
    }
}
```

Example (cont)



```
private static void fileData(File f) {
    System.out.println(
        "Absolute path: " + f.getAbsolutePath() +
        "\n Can read: " + f.canRead() +
        "\n Can write: " + f.canWrite() +
        "\n getName: " + f.getName() +
        "\n getParent: " + f.getParent() +
        "\n getPath: " + f.getPath() +
        "\n length: " + f.length() +
        "\n lastModified: " + f.lastModified());
    if(f.isFile())
        System.out.println("it's a file");
    else if(f.isDirectory())
        System.out.println("it's a directory");
}
```

Example (cont)



```
public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
        if(args.length != 3) usage();
        File
            old = new File(args[1]),
            rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);
        return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
        count++;
        del = true;
    }
}
```

Example (cont)



```
for( ; count < args.length; count++) {
    File f = new File(args[count]);
    if(f.exists()) {
        System.out.println(f + " exists");
        if(del) {
            System.out.println("deleting..." + f);
            f.delete();
        }
    }
    else { // Doesn't exist
        if(!del) {
            f.mkdirs();
            System.out.println("created " + f);
        }
    }
    fileData(f);
}
}
```

Example (explanation)

- In `fileData()` you can see the various **file investigation methods** put to use to display information about the file or directory path.
- `renameTo()` allows you to **rename (or move)** a file to an entirely new path represented by the argument, which is another **File** object.
 - This also works with directories of any length.
- you can **make a directory path** of any complexity because `mkdirs()` will do all the work for you.
 - the `-d` flag in Java 1.0 reports that the directory is deleted but it's still there; in Java 1.1 the directory is actually deleted.

Typical Uses of IO Streams

```
//: IOStreamDemo.java
// Typical IO Stream Configurations
import java.io.*;
public class IOStreamDemo {
    public static void main(String[] args) {
        try {
            // 1. Buffered input file
            DataInputStream in =
                new DataInputStream(
                    new BufferedInputStream(
                        new FileInputStream(args[0])));
            String s, s2 = new String();
            while((s = in.readLine()) != null)
                s2 += s + "\n";
            in.close();
        }
    }
}
```

Typical Uses of IO Streams

```
// 2. Input from memory
```

```
StringBufferInputStream in2 =  
    new StringBufferInputStream(s2);
```

```
int c;
```

```
while((c = in2.read()) != -1)
```

```
    System.out.print((char)c);
```

```
// 3. Formatted memory input
```

```
try {
```

```
    DataInputStream in3 =
```

```
        new DataInputStream(  
            new StringBufferInputStream(s2));
```

```
            while(true)
```

```
                System.out.print((char)in3.readByte());
```

```
                } catch(EOFException e) {
```

```
                    System.out.println(  
                        "End of stream encountered");
```

```
                    }
```

```
                    }
```

```
                }
```

Typical Uses of IO Streams

```
// 4. Line numbering & file output
try {
    LineNumberInputStream li =
        new LineNumberInputStream(
            new StringBufferInputStream(s2));
    DataInputStream in4 =
        new DataInputStream(li);
    PrintStream out1 =
        new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream(
                    "IODemo.out")));
    while((s = in4.readLine()) != null )
        out1.println(
            "Line " + li.getLineNumber() + s);
    out1.close(); // finalize() not reliable!
} catch(EOFException e) {
    System.out.println(
        "End of stream encountered");
}
```

Typical Uses of IO Streams

```
// 5. Storing & recovering data
try {
    DataOutputStream out2 =
        new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
    out2.writeBytes(
        "Here's the value of pi: \n");
    out2.writeDouble(3.14159);
    out2.close();
    DataInputStream in5 =
        new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("Data.txt")));
    System.out.println(in5.readLine());
    System.out.println(in5.readDouble());
} catch (EOFException e) {
    System.out.println(
        "End of stream encountered");
}
```

Typical Uses of IO Streams

```
// 6. Reading/writing random access files
RandomAccessFile rf =
    new RandomAccessFile("rtest.dat", "rw");
for(int i = 0; i < 10; i++)
    rf.writeDouble(i*1.414);
rf.close();

rf =
    new RandomAccessFile("rtest.dat", "rw");
rf.seek(5*8);
rf.writeDouble(47.0001);
rf.close();

rf =
    new RandomAccessFile("rtest.dat", "r");
for(int i = 0; i < 10; i++)
    System.out.println(
        "Value " + i + ": " +
        rf.readDouble());
rf.close();
```

Typical Uses of IO Streams



// 7. File input shorthand

```
InFile in6 = new InFile(args[0]);
String s3 = new String();
System.out.println(
    "First line in file: " +
    in6.readLine());
in6.close();
```

// 8. Formatted file output shorthand

```
PrintFile out3 = new PrintFile("Data2.txt");
out3.print("Test of PrintFile");
out3.close();
```

Typical Uses of IO Streams

// 9. Data file output shorthand

```
OutFile out4 = new OutFile("Data3.txt");
out4.writeBytes("Test of outDataFile\n\r");
out4.writeChars("Test of outDataFile\n\r");
out4.close();
```

```
} catch(FileNotFoundException e) {
    System.out.println(
        "File Not Found:" + args[0]);
} catch(IOException e) {
    System.out.println("IO Exception");
}
```

```
}
}
```

1. Buffered Input File

- To open a file for input, you use a **FileInputStream** with a **String** or a **File** object as the file name.
- For speed, you'll want that file to be buffered so you give the resulting handle to the constructor for a **BufferedInputStream**.
- To read input in a formatted fashion, you give that resulting handle to the constructor for a **DataInputStream**, which is your final object and the interface you read from.
- In this example, only the **readLine()** method is used, but of course any of the **DataInputStream** methods are available.
- When you reach the end of the file, **readLine()** returns **null** so that is used to break out of the **while** loop.
- The **String s2** is used to accumulate the entire contents of the file (including newlines that must be added since **readLine()** strips them off).
- Finally, **close()** is called to close the file.

2. Input From Memory

- o This piece takes the **String s2** that now contains the entire contents of the file and uses it to create a **StringBufferInputStream**. (A **String**, not a **StringBuffer**, is required as the constructor argument.)
- o Then **read()** is used to read each character one at a time and send it out to the console.
- o Note that **read()** returns the next byte as an **int** and thus it must be cast to a **char** to print properly.

3. Formatted Memory Input

- The interface for **StringBufferInputStream** is limited, so you usually enhance it by wrapping it inside a **DataInputStream**.
- However, if you choose to read the characters out a byte at a time using **readByte()**, any value is valid so the return value cannot be used to detect the end of input.
- Instead, you can use the **available()** method to find out how many more characters are available.

4. Line Numbering and File Output

- This example shows the use of the **LineNumberInputStream** to keep track of the input line numbers.
- Here, you cannot simply gang all the constructors together, since you have to keep a handle to the **LineNumberInputStream**.
- Thus, li holds the handle to the **LineNumberInputStream**, which is then used to create a **DataInputStream** for easy reading.

4. Line Numbering and File Output

- o This example also shows how to write formatted data to a file.
- o First, a **FileOutputStream** is created to connect to the file.
- o This is made a **BufferedOutputStream**, which is what you'll virtually always want to do, but you're forced to do it explicitly.
- o One of the methods that indicates when a **DataInputStream** is exhausted is **readLine()**, which returns **null** when there are no more strings to read. Each line is printed to the file along with its line number, which is acquired through `li`.

5. Storing and Recovering Data



- o A **PrintStream** formats data so it's readable by a human.
- o To output data so that it can be recovered by another stream, you use a **DataOutputStream** to write the data and a **DataInputStream** to recover the data.

6. Reading and Writing Random Access Files

- As previously noted, the **RandomAccessFile** is almost totally isolated from the rest of the IO hierarchy
- So you cannot combine it with any of the aspects of the **InputStream** and **OutputStream** subclasses.
- you can use **RandomAccessFile** to only open a file.
- The one option you have is in the second constructor argument: you can open a **RandomAccessFile** to read ("r") or read and write ("rw").
- Using a **RandomAccessFile** is like using a **combined DataInputStream and DataOutputStream**. In addition, you can see that **seek()** is used to move about in the file and change one of the values.

7. File Input Shorthand

- o The creation of an object that reads a file from a buffered `DataInputStream` can be encapsulated into a class called `InFile`:

```
//: InFile.java  
// Shorthand class for opening an input file
```

```
import java.io.*;  
  
public class InFile extends DataInputStream {  
    public InFile(String filename)  
        throws FileNotFoundException {  
        super(  
            new BufferedInputStream(  
                new FileInputStream(filename)));  
    }  
    public InFile(File file)  
        throws FileNotFoundException {  
        this(file.getPath());  
    }  
}
```

8. Formatted File Output Shorthand

- o The same kind of approach can be taken to create a **PrintStream** that writes to a buffered file.

```
//: PrintFile.java
// Shorthand class for opening an output file
// for human-readable output.
import java.io.*;
```

```
public class PrintFile extends PrintStream {
    public PrintFile(String filename)
        throws IOException {
        super(
            new BufferedOutputStream(
                new FileOutputStream(filename)));
    }
    public PrintFile(File file)
        throws IOException {
        this(file.getPath());
    }
}
```

9. Data File Output Shorthand

- o Finally, the same kind of shorthand can create a buffered output file for data storage (as opposed to human-readable storage):

```
//: OutFile.java
// Shorthand class for opening an output file
// for data storage.
import java.io.*;

public class OutFile extends DataOutputStream {
    public OutFile(String filename)
        throws IOException {
        super(
            new BufferedOutputStream(
                new FileOutputStream(filename)));
    }
    public OutFile(File file)
        throws IOException {
        this(file.getPath());
    }
}
```

Reading From Standard Input

- o Following the approach pioneered in Unix of "standard input," "standard output," and "standard error output," Java has **System.in**, **System.out**, and **System.err**.
- o **System.out**, which is already pre-wrapped as a **PrintStream** object.
- o **System.err** is likewise a **PrintStream**,
- o but **System.in** is a raw **InputStream**, with no wrapping.
- o This means that while you can use **System.out** and **System.err** right away, **System.in** must be wrapped before you can read from it.

Reading From Standard Input

- o Typically, you read input a line at a time using `readLine()`, so you'll want to wrap `System.in` in an `DataInputStream`.

```
//: Echo.java
// How to read from standard input
import java.io.*;
public class Echo {
    public static void main(String[] args) {
        DataInputStream in =
            new DataInputStream(
                new BufferedInputStream(System.in));
        String s;
        try {
            while((s = in.readLine()).length() != 0)
                System.out.println(s);
            // An empty line terminates the program
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Java 1.1 IO Streams



- New classes have been put into the old hierarchy, so it's obvious that Sun is not abandoning the old streams.
- There are times when you're supposed to use classes in the old hierarchy *in combination* with classes in the new hierarchy and to accomplish this there are "bridge" classes: **InputStreamReader** converts an **InputStream** to a **Reader** and **OutputStreamWriter** converts an **OutputStream** to a **Writer**.

Java 1.1 IO Streams



- As a result there are situations in which you have *more layers* of wrapping with the new IO stream library than with the old.
- The most important reason for adding the **Reader** and **Writer** hierarchies in Java 1.1 is for *internationalization*.
 - The old IO stream hierarchy supports only 8-bit byte streams and doesn't handle the 16-bit Unicode characters well.

Java 1.1 IO Streams



- In addition, the new libraries are designed for **faster operations** than the old.
- The most sensible approach to take is to *try* to use the **Reader** and **Writer** classes whenever you can, you'll discover the situations when you have to drop back into the old libraries because **your code won't compile**.

Java 1.1 IO Streams

Sources & Sinks: Java 1.0 class	Corresponding Java 1.1 class
InputStream	Reader converter: InputStreamReader
OutputStream	Writer converter: OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(no corresponding class)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

Modifying Stream Behaviour

Filters: Java 1.0 class	Corresponding Java 1.1 class
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (abstract class with no subclasses)
BufferedInputStream	BufferedReader (also has readLine())
BufferedOutputStream	BufferedWriter
DataInputStream	use DataInputStream (Except when you need to use readLine() , when you should use a BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer (use constructor that takes a Reader instead)
PushBackInputStream	PushBackReader

Unchanged Classes



- Some of the classes have **no changes**. These are:
 - **File**
 - **RandomAccessFile**
 - **SequenceInputStream**
 - The **DataOutputStream**, in particular, is used without change, so for storing and retrieving data in a transportable format you're forced to stay in the **InputStream** and **OutputStream** hierarchies.

Example



- o To see the effect of the new classes, let's look at the appropriate portion of the **IOStreamDemo.java** example modified to use the **Reader** and **Writer** classes:

```
//: NewIODemo.java
```

```
// Java 1.1 IO typical usage
```

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

```
public class IO11Demo {  
    public static void main(String[] args) {  
        try {
```

Example (cont)

```
// 1. Reading input by lines:
```

```
BufferedReader in =  
    new BufferedReader(  
        new FileReader(args[0]));  
String s, s2 = new String();  
while((s = in.readLine()) != null)  
    s2 += s + "\n";  
in.close();
```

```
// 1b. Reading standard input:
```

```
BufferedReader stdin =  
    new BufferedReader(  
        new InputStreamReader(System.in));  
System.out.print("Enter a line:");  
System.out.println(stdin.readLine());
```

Example (cont)

```
// 2. Input from memory
StringReader in2 = new StringReader(s2);
int c;
while((c = in2.read()) != -1)
    System.out.print((char)c);
// 3. Formatted memory input
try {
    DataInputStream in3 =
        new DataInputStream(
            // Oops: must use deprecated class:
            new StringBufferInputStream(s2));
    while(true)
        System.out.print((char)in3.readByte());
} catch(EOFException e) {
    System.out.println("End of stream");
}
```

Example (cont)

```
// 4. Line numbering & file output
try {
    LineNumberReader li =
        new LineNumberReader(
            new StringReader(s2));
    BufferedReader in4 =
        new BufferedReader(li);
    PrintWriter out1 =
        new PrintWriter(
            new BufferedWriter(
                new FileWriter("IODemo.out")));
    while((s = in4.readLine()) != null )
        out1.println(
            "Line " + li.getLineNumber() + s);
    out1.close();
} catch(EOFException e) {
    System.out.println("End of stream");
}
```

Example (cont)

```
// 5. Storing & recovering data
try {
    DataOutputStream out2 =
        new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
    out2.writeDouble(3.14159);
    out2.writeBytes("That was pi");
    out2.close();
    DataInputStream in5 =
        new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("Data.txt")));
    BufferedReader in5br =
        new BufferedReader(
            new InputStreamReader(in5));
    // Must use DataInputStream for data:
    System.out.println(in5.readDouble());
    // Can now use the "proper" readLine():
    System.out.println(in5br.readLine());
} catch (EOFException e) {
    System.out.println("End of stream");
}
```

Example (cont)



```
// 6. Reading and writing random access
// files is the same as before.
// (not repeated here)
```

```
} catch(FileNotFoundException e) {
    System.out.println(
        "File Not Found:" + args[1]);
} catch(IOException e) {
    System.out.println("IO Exception");
}
}
```

Example (explanations)

- There are some important differences:
 - First of all, since random access files have not changed, **section 6** is not repeated.
 - **Section 1** shrinks a bit because if all you're doing is reading line input you need only to wrap a **BufferedReader** around a **FileReader**.
 - **Section 1b** shows the new way to wrap **System.in** for reading console input, and this expands because **System.in** is an **InputStream** and **BufferedReader** needs a **Reader** argument, so **InputStreamReader** is brought in to perform the translation.

Example (explanations)

- In **section 2** you can see that if you have a **String** and want to read from it you just use a **StringReader** instead of a **StringBufferInputStream** and the rest of the code is identical.
- **Section 3** shows a bug in the design of the new IO stream library.
 - ✓ If you have a **String** and you want to read from it, you're *not* supposed to use a **StringBufferInputStream** any more.
 - ✓ When you compile code involving a **StringBufferInputStream** constructor, you get a deprecation message telling you to not use it.
 - ✓ Instead, you're supposed to use a **StringReader**.
 - ✓ However, if you want to do formatted memory input as in section 3, you're forced to use a **DataInputStream** whose constructor requires an **InputStream** argument.
 - ✓ So you have no choice but to use the deprecated **StringBufferInputStream** class. The compiler will give you a deprecation message but there's nothing you can do about it.

Example (explanations)

- **Section 4** is a reasonably straightforward translation from the old streams to the new, with no surprises.
- In **section 5**, you're forced to use all the old streams classes because **DataOutputStream** and **DataInputStream** require them and there are no alternatives.
 - ✓ However, you don't get any deprecation messages at compile time.
 - ✓ If you compare section 5 with that section in **IOStreamDemo.java**, you'll notice that in *this* version, the data is written *before* the text.
 - ✓ That's because a bug was introduced in Java 1.1

Redirecting Standard IO



- o Java 1.1 has added methods in class **System** that allow you to redirect the standard input, output, and error IO streams using simple static method calls:
 - **setIn(InputStream)**
 - **setOut(PrintStream)**
 - **setErr(PrintStream)**

Example

```
//: Redirecting.java
// Demonstrates the use of redirection for
// standard IO in Java 1.1
import java.io.*;

class Redirecting {
    public static void main(String[] args) {
        try {
            BufferedInputStream in=new BufferedInputStream(new FileInputStream(
                "Redirecting.java"));
            // Produces deprecation message:
            PrintStream out=new PrintStream(new BufferedOutputStream(new
            FileOutputStream("test.out")));
            System.setIn(in);
            System.setOut(out);
            System.setErr(out);

            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
            String s;
            while((s = br.readLine()) != null)
                System.out.println(s);
            out.close(); // Remember this!
        } catch(IOException e) {
            e.printStackTrace();
        } } }
```

Object Serialization



- o Java 1.1 has added an interesting feature called *object serialization* that allows you to take any object that implements the **Serializable** interface and turn it into a sequence of bytes that can later be restored fully into the original object.
- o This is even **true across a network**, which means that the serialization mechanism automatically compensates for differences in operating systems.

Object Serialization

- Object serialization allows you to implement *lightweight persistence*.
- **Persistence**: the object lives *in between* invocations of the program.
- You must **explicitly serialize and de-serialize** the objects in your program.
- Object serialization was added to the language to support two major features:
 - Java 1.1's *remote method invocation* (RMI) allows objects that live on other machines to behave as if they live on your machine.
 - Object serialization is also necessary for **Java Beans**, introduced in Java 1.1.

Object Serialization

- Serializing an object is quite simple, **as long as the object implements the `Serializable`** interface.
- To serialize an object, you create some sort of **`OutputStream`** object and then wrap it inside an **`ObjectOutputStream`** object.
- At this point you need only call **`writeObject()`** and your object is serialized and sent to the **`OutputStream`**.
- To reverse the process, you wrap an **`InputStream`** inside an **`ObjectInputStream`** and call **`readObject()`**.
 - What comes back is, as usual, **a handle to an upcast `Object`**, so you must downcast to set things straight.

Object Serialization



- o A particularly clever aspect of object serialization is that it not only saves an image of your object but it also **follows all the handles contained in your object and saves those objects**, and follows all the handles in each of those objects, etc.

Serialization Example



```
//: Worm.java
// Demonstrates object serialization in Java 1.1
import java.io.*;

class Data implements Serializable {
    private int i;
    Data(int x) { i = x; }
    public String toString() {
        return Integer.toString(i);
    }
}
```

Serialization Example (cont)

```
public class Worm implements Serializable {
    // Generate a random int value:
    private static int r() {
        return (int)(Math.random() * 10);
    }
    private Data[] d = {
        new Data(r()), new Data(r()), new Data(r())
    };
    private Worm next;
    private char c;
    // Value of i == number of segments
    Worm(int i, char x) {
        System.out.println(" Worm constructor: " + i);
        c = x;
        if(--i > 0)
            next = new Worm(i, (char)(x + 1));
    }
}
```

Serialization Example (cont)



```
Worm() {
    System.out.println("Default constructor");
}
public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++)
        s += d[i].toString();
    s += ")";
    if(next != null)
        s += next.toString();
    return s;
}
```

Serialization Example (cont)

```
public static void main(String[] args) {
    Worm w = new Worm(6, 'a');
    System.out.println("w = " + w);
    try {
        ObjectOutputStream out=new ObjectOutputStream(
            new FileOutputStream("worm.out"));
        out.writeObject("Worm storage");
        out.writeObject(w);
        out.close(); // Also flushes output
        ObjectInputStream in =
            new ObjectInputStream(
                new FileInputStream("worm.out"));
        String s = (String)in.readObject();
        Worm w2 = (Worm)in.readObject();
        System.out.println(s + ", w2 = " + w2);
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

Serialization Example (cont)

```
try {
    ByteArrayOutputStream bout =
        new ByteArrayOutputStream();
    ObjectOutputStream out =
        new ObjectOutputStream(bout);
    out.writeObject("Worm storage");
    out.writeObject(w);
    out.flush();
    ObjectInputStream in = new ObjectInputStream(
        new ByteArrayInputStream(bout.toByteArray()));
    String s = (String)in.readObject();
    Worm w3 = (Worm)in.readObject();
    System.out.println(s + ", w3 = " + w3);
} catch (Exception e) {
    e.printStackTrace();
}
}
```

Example output



Worm constructor: 6

Worm constructor: 5

Worm constructor: 4

Worm constructor: 3

Worm constructor: 2

Worm constructor: 1

w = :a(262):b(100):c(396):d(480):e(316):f(398)

Worm storage, w2 =

:a(262):b(100):c(396):d(480):e(316):f(398)

Worm storage, w3 =

:a(262):b(100):c(396):d(480):e(316):f(398)

Object Serialization

- You can see that the deserialized object really does contain all of the links that were in the original object.
- Note that **no constructor**, not even the default constructor, is **called in the process of deserializing** a **Serializable** object. The entire object is restored by recovering data from the **InputStream**.
- Object serialization is another Java 1.1 feature that is **not part of the new Reader and Writer** hierarchies, but instead uses the old **InputStream** and **OutputStream** hierarchies. Thus you might encounter situations in which you're forced to mix the two hierarchies.