10

Using Stream for I/O

Objectives

After completing this lesson, you should be able to do the following:

- Use streams for input and output of byte data
- Use streams for input and output of character data
- Generate formatted output
- Handle remote I/O
- Use object streams to support the I/O of objects
- Handle exceptions when dealing with I/O

Streams

- Anything that is read from or written to is a stream.
- Examples:
  - Console
  - File
  - Pipes
  - Network
  - Memory
- Examples of things that can be read or written:
  - Characters
  - (Serializable) Java objects
  - Sound, images

Sets of I/O Classes

- All modern I/O is stream-based.
- The `java.io` package contains a collection of classes that support reading and writing from streams.
- A program needs to import the `java.io` package to use these classes.
- The stream classes are divided into two class hierarchies based on the data type on which they operate:
  - Byte streams
  - Character streams

How to Do I/O

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

- *Open* the stream.
- *Use* the stream (read, write, or both).
- *Close* the stream.

Why Java I/O Is Hard

- Java is very powerful, with an overwhelming number of options.
- Any given kind of I/O is not particularly difficult.
- The trick is to find your way through the maze of possibilities.

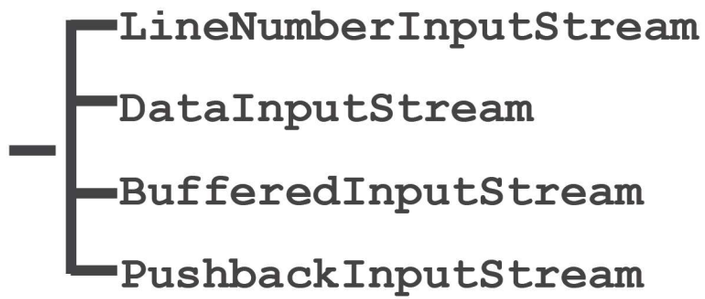
Byte I/O Streams

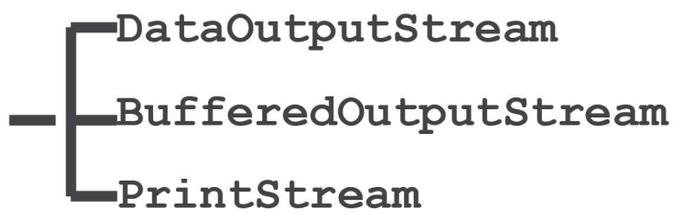
InputStream

- FileInputStream
- PipedInputStream
- FilterInputStream
- …

OutputStream

- FileOutputStream
- PipedOutputStream
- FilterOutputStream
- …





The methods in the `InputStream` class include:

- `read()`
- `read(byte b[])`
- `read(byte b[], int off, int len)`
- `skip(long n)`
- `available()`
- `close()`
- `mark(int readlimit)`
- `reset()`
- `markSupported()`

The methods in the `OutputStream` class include:

- `write(int b)`
- `write(byte b[])`
- `write(byte b[],int off int len)`
- `flush()`
- `close()`

Using Byte Streams

- Byte streams should only be used for the most primitive I/O.
- They are important because all other streams are built on byte streams.
- There are many byte stream classes.
- `FileInputStream` and `FileOutputStream` are examples of file I/O byte streams.

Using Character Streams

- **Reader**
  - **BufferedReader** `LineNumberInputStream`
  - **CharArrayReader**
  - **InputStreamReader** `FileReader`
  - ...
- **Writer**
  - **BufferedWriter**
  - **CharArrayWriter**
  - **OutputStreamWriter** `FileWriter`
  - ...

Using Character Streams

- The Java platform stores character values using Unicode conventions.
- Character stream I/O translates this internal format to and from the local character set.
- Character streams allow for the internationalizing of applications without extensive recoding.
- Character streams are often “wrappers” for byte streams.

The `InputStreamReader` Class

- Is a character input stream that uses a byte input stream as its data source
- Reads bytes from a specified `InputStream` and translates them into Unicode characters, according to a particular platform- and locale-dependent character encoding
- Has a `getEncoding()` method that returns the name of the encoding being used to convert bytes to characters

The `OutputStreamWriter` Class

- Is a character output stream that uses a byte output stream as the destination for its data
- Translates characters into bytes according to a particular locale and/or platform-specific character encoding, and writes those bytes to a specified `OutputStream`
- Supports all the usual `Writer` methods
- Has a `getEncoding()` method that returns the name of the encoding being used to convert characters to bytes

The Basics: Standard Output

Understanding `System.out.println()`:

- There is no package called `System` with a class named `out` and a `println()` method.
- `System` is a class in the `java.lang` package.
- `out` is a `public final static` (class) variable.
  - Declared as a `PrintStream` object reference
- `println()` is an overloaded method of the `PrintStream` class.
  - `PrintStream` is a `FilterOutputStream` that subclasses `OutputStream`.

PrintStream and PrintWriter

- Stream objects that implement formatting are instances of either `PrintStream` or `PrintWriter`.
- `PrintStream`:
  - Is a byte stream class and a subclass of `(Filter)OutputStream`
  - Converts Unicode to environment byte encoding
  - Terminates lines in a platform-independent way
  - Flushes the output stream
- `PrintWriter`:
  - Is a character stream class and a subclass of `Writer`
  - Implements all of the print methods found in `PrintStream`
  - Only flushes when a `println()` method is invoked

Formatted Output

`Printf`:

- Is used for generating formatted output
- Provides support for:
  - Layout justification and alignment
  - Common formats for numeric, string, and date-time data
  - Locale-specific output
- Is inspired by `printf` in C

Format Specifiers

- A format specifier specifies how an item should be displayed.
- An item may be a numeric value, a character, a Boolean value or a string.
- Each format specifier begins with a percent sign.

The Basics: Standard Input

- Standard input is represented by `System.in`.
- `in` is an `InputStream` object that provides limited reading ability through the `read()` method.
- `in` does not enable you to:
  - Read one line at a time
  - Read a Java primitive, such as a double
- `in` is useful when you want to "pause" your program. `try {System.in.read();} catch ...`

Scanner API

- Was introduced in JDK 5.0
- Provides improved input functionality for reading data from the system console or any data stream
- Can be used to convert text into primitives or strings
- Offers a way to conduct expression-based searches on streams, file data, strings, and so on

Remote I/O

- Remote I/O is accomplished by sending data across a network connection.
- Java provides several networking classes in the package `java.net`.
- You specify a URL for a remote file:
  - Open an `InputStream` to read it.
  - Send HTTP requests via HTTP classes.
  - Access sockets via `Socket` and `ServerSocket`.
- Communication is achieved using `InputStream` and `OutputStream`.
- For the Java I/O model, it is not important where the data is coming from, or where you are sending it: you just `read()` and `write()`.

Data Streams

- Support binary I/O of values of primitive data types (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`) as well as string values
- Implement either the `DataInput` interface or the `DataOutput` interface

Object Streams

- Support I/O of objects
- Are implemented by the `ObjectInputStream` and `ObjectOutputStream` classes
- Implement all the primitive data I/O methods covered in data streams
- Can contain a mixture of primitive and object values

Object Serialization

Serialization is a lightweight persistence mechanism for saving and restoring streams of bytes containing primitives and objects.

A class indicates that its instances can be serialized by:

- Implementing the `java.io.Serializable` or `java.io.Externalizable` interfaces
- Ensuring that all its fields are serializable, including other referenced objects
- Using the `transient` modifier to prevent fields from being saved and restored

Example of implementing `java.io.Serializable`

- Mark fields with the `transient` modifier to prevent them from being saved (that is, to protect the information):

```
import java.io.Serializable;
public class Member implements Serializable {
  private int id;
  private String name;
  private transient String password;
  …
}
```

- Write objects with `java.io.ObjectOutputStream`.
- Read objects with `java.io.ObjectInputStream`.

`IOException` Class

- Many things can go wrong when dealing with I/O:
  - The requested input file does not exist.
  - The input file has invalid data.
  - The output file is unavailable.
  - And so on
- The `IOException` class is used by many methods in `java.io` to signal exceptional conditions.
- Some extended classes of `IOException` signal specific problems.
- Most problems are signalled by an `IOException` object with a string describing the specific error encountered.

Summary

In this lesson, you should have learned how to:

- Use streams for input and output of byte data
- Use streams for input and output of character data
- Handle remote I/O
- Use object streams to support the I/O of objects
- Handle exceptions when dealing with I/O

Practice : Overview

This practice covers the following topics:

- Writing a file containing customer information
- Reading the file using `FileInputStream`, and then printing it out as byte values
- Reading the same file using `InputStreamReader`, and then printing output as characters
- Using `Scanner` to read and output the file
- Using object serialization to save and restore the application data