# Introduction to Java I/O

Presented by developerWorks, your source for great tutorials

**ibm.com/developerWorks**

---

## Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.


1. Tutorial tips 2
2. An overview of the java.io package 3
3. java.io class overview 8
4. Sources and sinks 13
5. Files 19
6. Buffering 25
7. Filtering 30
8. Checksumming 34
9. Inflating and deflating 38
10. Data I/O 44
11. Object serialization 49
12. Tokenizing 54
13. Lab 58
14. Wrapup 59

# Section 1. Tutorial tips

## Should I take this tutorial?

This tutorial is an overview of Java I/O and all the classes in the `java.io` package. We journey through the maze of the `java.io` package, examining I/O classes, methods, and various techniques for handling I/O in your Java code.

This tutorial assumes you have a basic knowledge of I/O, including `InputStream` and `OutputStream`. If you have training and experience in Java programmming, take this course to add to your knowledge. If you do not have Java programming experience, we suggest you take *Introduction to Java for COBOL Programmers* , *Introduction to Java for C or C++ Programmers* , or other introductory Java courses available on the Web.

In this tutorial, we provide examples of code to show details of Java I/O. If you want to compile and run the same code and you do not have a Java compiler, you can download the Java Development Kit (JDK) from Sun Microsystems. (See Setup on page 58.) You may use either the JDK 1.1 or JDK 1.2 (also known as Java 2).

# Section 2. An overview of the java.io package

## Introduction

This section introduces the `java.io` package.

Here are some basic points about I/O:

* Data in files on your system is called *persistent* data because it persists after the program runs.
* Files are created through *streams* in Java code.
* A stream is a linear, sequential flow of bytes of input or output data.
* Streams are written to the file system to create files.
* Streams can also be transferred over the Internet.
* Three streams are created for us automatically:
  - `System.out` - standard output stream
  - `System.in` - standard input stream
  - `System.err` - standard error

* Input/output on the local file system using applets is dependent on the browser's security manager. Typically, I/O is not done using applets. On the other hand, stand-alone applications have no security manager by default unless the developer has added that functionality.

---

## Basic input and output classes

The `java.io` package contains a fairly large number of classes that deal with Java input and output. Most of the classes consist of:

* Byte streams that are subclasses of `InputStream` or `OutputStream`
* Character streams that are subclasses of `Reader` and `Writer`

The `Reader` and `Writer` classes read and write 16-bit Unicode characters. `InputStream` reads 8-bit bytes, while `OutputStream` writes 8-bit bytes. As their class name suggests, `ObjectInputStream` and `ObjectOutputStream` transmit entire objects. `ObjectInputStream` reads objects; `ObjectOutputStream` writes objects.

Unicode is an international standard character encoding that is capable of representing most of the world's written languages. In Unicode, two bytes make a character.

Using the 16-bit Unicode character streams makes it easier to internationalize your code. As a result, the software is not dependent on one single encoding.

---

## What to use

There are a number of different questions to consider when dealing with the `java.io` package:

* What is your format: text or binary?
* Do you want random access capability?
* Are you dealing with objects or non-objects?
* What are your sources and sinks for data?
* Do you need to use filtering?

---

## Text or binary

What's your format for storing or transmitting data? Will you be using text or binary data?

* If you use binary data, such as integers or doubles, then use the `InputStream` and `OutputStream` classes.
* If you are using text data, then the `Reader` and `Writer` classes are right.

---

## Random access

Do you want random access to records? Random access allows you to go anywhere within a file and be able to treat the file as if it were a collection of records.

The `RandomAccessFile` class permits random access. The data is stored in binary format. Using random access files improves performance and efficiency.

---

## Object or non-object

Are you inputting or outputting the attributes of an object? If the data itself is an object, then use the `ObjectInputStream` and `ObjectOutputStream` classes.

---

## Sources and sinks for data

What is the source of your data? What will be consuming your output data, that is, acting as a sink? You can input or output your data in a number of ways: sockets, files, strings, and arrays of characters.

Any of these can be a source for an `InputStream` or `Reader` or a sink for an `OutputStream` or `Writer`.

---

## Filtering

Do you need filtering for your data? There are a couple ways to filter data.

*Buffering* is one filtering method. Instead of going back to the operating system for each byte, you can use an object to provide a buffer.

*Checksumming* is another filtering method. As you are reading or writing a stream, you might want to compute a checksum on it. A checksum is a value you can use later on to make sure the stream was transmitted properly.

We cover the concepts and details of filtering in Filtering on page 30; and the details of checksumming are in Checksumming on page 34.

---

# Storing data records

A data record is a collection of more than one element, such as names or addresses. There are three ways to store data records:

* Use *delimited records*, such as a mail-merge style record, to store values. On output, data values are converted to strings separated by delimiters such as the tab character, and ending with a new-line character. To read the data back into Java code, the entire line is read and then broken up using the `StringTokenizer` class.
* Use *fixed size records* to store data records. Use the `RandomAccessFile` class to store one or more records. Use the `seek()` method to find a particular record. If you choose this option to store data records, you must ensure strings are set to a fixed maximum size.
* Alternatively, you can use *variable length records* if you use an auxiliary file to store the lengths of each record. Use object streams to store data records. If object streams are used, no skipping around is permitted, and all objects are written to a file in a sequential manner.

---

# Creating streams: Example code

Here is an example of how to create a stream that reads and writes characters using a TCP/IP socket as the sink and source. The classes themselves are explained later.

First, we create a TCP/IP socket object that is connected to www.ibm.com and port 80. This is the Web server port. The method `getInputStream()` in the `Socket` class returns an `InputStream`, which represents byte-by-byte reading of the socket. The `InputStream` is used to create an `InputStreamReader`, which transforms the bytes read from the socket into characters. A `BufferedReader` class is created, which reads from the `InputStreamReader` and buffers the characters into its own internal buffer. The object named `in` then reads characters from that buffer, but the ultimate source of the characters is the Web server at www.ibm.com.

On the other hand, the `getOutputStream()` method of the `Socket` class returns a reference to an `OutputStream`, which writes a byte at a time. The `PrinterWriter`

constructor uses that `OutputStream` as the sink for the characters it writes out. When a character is written to the object named `out`, it is ultimately sent to the Web server at www.ibm.com.

This example treats the data stream as character data:

```
Socket a_socket = new Socket(www.ibm.com, 80);
   InputStreamReader isr = new InputStreamReader(
      a_socket.getInputStream());
   BufferedReader in = new BufferedReader (isr);
   PrinterWriter out = new PrinterWriter(
      a_socket.getOutputStream());
```

A second way to use streams is to use them to transmit binary data. We create a TCP/IP socket object that is connected to www.ibm.com and port 100. We construct a `DataInputStream` using the `InputStream` returned by `getInputStream()` and a `DataOutputStream` using the `OutputStream` returned by `getOutputStream()`. We can send and receive integers or doubles or other binary data over these two streams.

This example treats the data stream as binary data:

```
Socket socket_data = new Socket(www.ibm.com, 100);
   DataInputStream in_data = new DataInputStream(
      socket_data.getInputStream());
   DataOutputStream out_data = new DataOutputStream(
      socket_data.getOutputStream());
```

---

# Exceptions

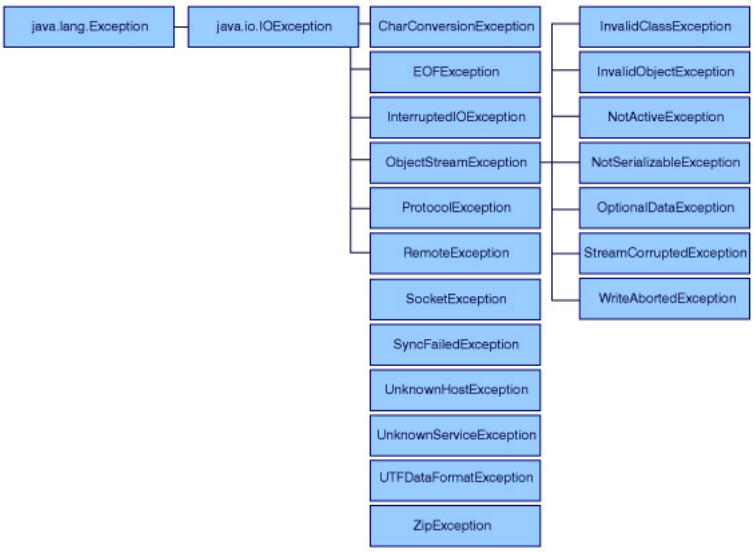
Almost every input or output method throws an exception. Therefore, any time you do I/O, you need to catch exceptions. There is a large hierarchy of I/O exceptions derived from `IOException`. Typically you can just catch `IOException`, which catches all the derived exceptions. However, some exceptions thrown by I/O methods are not in the `IOException` hierarchy. One such exception is the `java.util.zip.DataFormatException.` This exception is thrown when invalid or corrupt data is found while data being read from a zip file is being uncompressed. `java.util.zip.DataFormatException` has to be caught explicitly because it is not in the `IOException` hierarchy.

Remember, exceptions in Java code are thrown when something unexpected happens.

---

# List of exceptions

This figure shows a list of exceptions:




Let's look at a few examples:

* `EOFException` signals when you reach an end-of-file unexpectedly.
* `UnknownHostException` signals that you are trying to connect to a remote host that does not exist.
* `ZipException` signals that an error has occurred in reading or writing a zip file.

Typically, an `IOException` is caught in the `try` block, and the value of the `toString()` method of the `Object` class in the `IOException` is printed out as a minimum. You should handle the exception in a manner appropriate to the application.

# Section 3. java.io class overview

## Introduction

This section introduces the basic organization of the `java.io` classes, consisting of:

* Input and output streams
* Readers and writers
* Data and object I/O streams

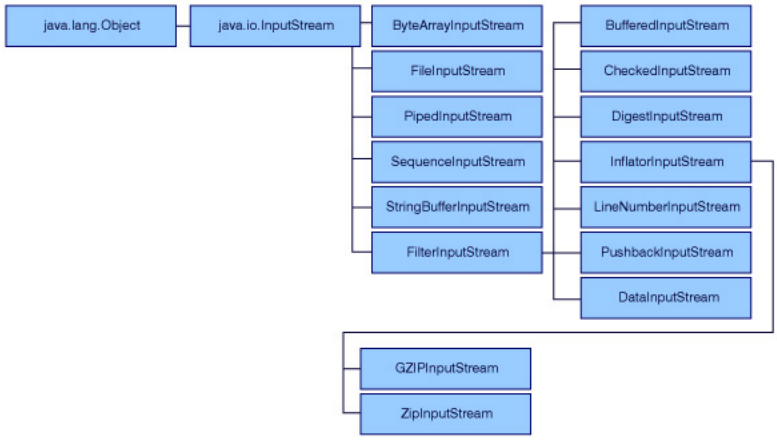
---

## Input stream classes

In the `InputStream` class, bytes can be read from three different sources:

* An array of bytes
* A file
* A pipe

Sources, such as `ByteArrayInputStream` and `FilterInputStream`, subclass the `InputStream` class. The subclasses under `InflatorInputStream` and `FilterInputStream` are listed in the figure below.




---

## InputStream methods

Various methods are included in the `InputStream` class.

* `abstract int read()` reads a single byte, an array, or a subarray of bytes. It returns the bytes read, the number of bytes read, or -1 if end-of-file has been reached.
* `read()`, which takes the byte array, reads an array or a subarray of bytes and returns a -1 if the end-of-file has been reached.
* `skip()`, which takes `long`, skips a specified number of bytes of input and returns the

number of bytes actually skipped.
* `available()` returns the number of bytes that can be read without blocking. Both the input and output can block threads until the byte is read or written.
* `close()` closes the input stream to free up system resources.

---

## InputStream marking

Some, but not all, `InputStreams` support marking. Marking allows you to go back to a marked place in the stream like a bookmark for future reference. Remember, not all `InputStreams` support marking. To test if the stream supports the `mark()` and `reset()` methods, use the boolean `markSupported()` method.

The `mark()` method, which takes an `integer read_limit`, marks the current position in the input stream so that `reset()` can return to that position as long as no more than the specified number of bytes have been read between the `mark()` and `reset()`.

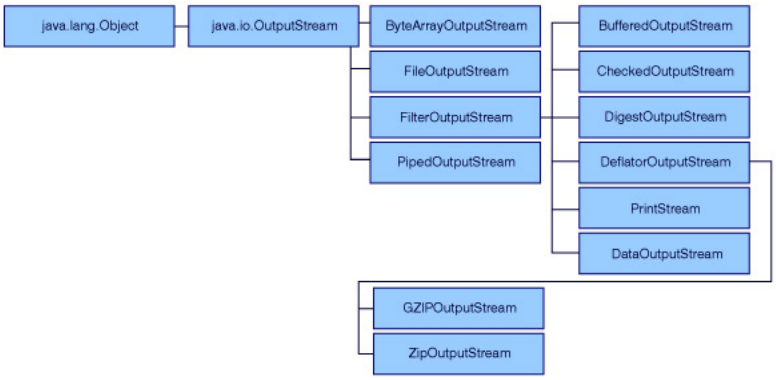
---

## OutputStream classes

Bytes can be written to three different types of sinks:

* An array of bytes
* A file
* A pipe

The following figure shows OutputStream classes.




Let's look at some examples of `OutputStream` classes. Before sending an `OutputStream` to its ultimate destination, you can filter it. For example, the `BufferedOutputStream` is a subclass of the `FilterOutputStream` that stores values to be written in a buffer and writes them out only when the buffer fills up.

`CheckedOutputStream` and `DigestOutputStream` are part of the `FilterOutputStream` class. They calculate checksums or message digests on the output.

`DeflatorOutputStream` writes to an `OutputStream` and creates a zip file. This class does compression on the fly.

The `PrintStream` class is also a subclass of `FilterOutputStream`, which implements a number of methods for displaying textual representation of Java primitive types. For example:

* println(long)
* println(int)
* println(float)
* println(char)

`DataOutputStream` implements the `DataOutput` interface. `DataOutput` defines the methods required for streams that can write Java primitive data types in a machine-independent binary format.

---

# OutputStream methods

The `OutputStream` class provides several methods:

* The abstract `void write()` method takes an integer byte and writes a single byte.
* The `void write()` method takes a byte array and writes an array or subarray of bytes.
* The `void flush()` method forces any buffered output to be written.
* The `void close()` method closes the stream and frees up system resources.

It is important to close your output files because sometimes the buffers do not get completely flushed and, as a consequence, are not complete.

---

# DataInput and DataOutput

The `DataInput` and `DataOutput` classes define the methods required for streams that can read Java primitive data types in a machine-independent binary format. They are implemented by `RandomAccessFile`.

The `ObjectInput` interface extends the `DataInput` interface and adds methods for deserializing objects and reading bytes and arrays of bytes. You can learn more about serialization in Object serialization on page 49.

The `ObjectOutputStream` class creates a stream of objects that can be deserialized by the `ObjectInputStream` class.

The `ObjectOutput` interface extends the `DataOutput` interface and adds methods for serializing objects and writing bytes and arrays of bytes.

`ObjectOutputStream` is used to serialize objects, arrays, and other values to a stream.

## What are Readers?

**read() API:**

```
int read()

read (char[] buffer,
  int offset, int length)
```

Readers are character-based input streams that read Unicode characters.

* `read()` reads a single character and returns a character read as an integer in the range from 0 to 65535 or a `-1` if the end of the stream is reached.
* `abstract read()` reads characters into a portion of an array (starting at offset up to length number of characters) and returns the number of characters read or `-1` if the end of the stream is reached.

## Character input streams

Let's take a look at the different character input streams in the `java.io` package.

* Strings
* Character arrays
* Pipes

`InputStreamReader` is a character input stream that uses a byte input stream as its data source and converts it into Unicode characters.

`LineNumberReader`, a subclass of `BufferedReader`, is a character input stream that keeps track of the number of lines of text that have been read from it.

`PushbackReader`, a subclass of `FilterReader`, is a character input stream that uses another input stream as its input source and adds the ability to push characters back onto the stream.

## What are Writers?

```
Writer() API:

void write(int character)

void write(char[] buffer,
  int offset, int length)
```

Writers are character-based output streams that write character bytes and turn Unicode into bytes. The base class includes these methods:

* The `void write()` method, which takes a character and writes single character in 16 low-order bits
* The `abstract void write()` method, which takes a character array and writes a portion of an array of characters

---

# Character output streams

Let's take a look at the different character output streams in the `java.io` package. There are several branches of this inheritance tree you can explore. Like `Readers`, any of the branches are available. Sinks for `Writer` output can be:

* Strings
* CharArray
* Pipes

`OutputStreamWriter` uses a byte output stream as the destination for its data.

`BufferedWriter` applies buffering to a character output stream, thus improving output efficiency by combining many small write requests into a single large request.

`FilterWriter` is an abstract class that acts as a superclass for character output streams. The streams filter the data written to them before writing it to some other character output stream.

`PrintWriter` is a character output stream that implements `print()` and `println()` methods that output textual representations of primitive values and objects.

# Section 4. Sources and sinks

## Introduction

This section focuses on sources and sinks. Code samples help you visualize how some of these classes work:

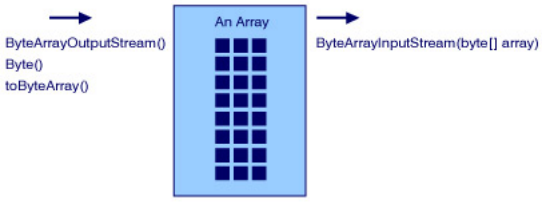
* Byte arrays
* Pipes
* Sequences
* Char arrays

---

## Byte arrays

Let's head toward byte array territory. Are bytes going to come from a byte array? When an object of a `ByteArrayInputStream` is created, it is passed an array. You can then read bytes from that array as an `InputStream`.

`ByteArrayOutputStream` writes to a byte array. Bytes that are written to the output stream are added to an internal array. You may be wondering how you get to the byte array. The `toByteArray()` method returns to you the byte array at any point in time. Also, a `toString()` method is available, which returns the byte array as a `String`. Therefore, the byte array can be either a source or a sink.




---

## Pipes in Java code

What is a pipe? A pipe is a means of communication between two threads of a Java program. One thread writes to a piped output stream and another thread reads from a piped input stream, which is connected to the piped output stream.

A pipe between two threads works like a pipe works between two processes in UNIX and MS-DOS. In those operating systems, the output from one process can be piped to the input of another process. For example, in MS-DOS, the command "dir | more" pipes the output of the "dir" command to the input of the "more" program.

---

## Pipe: Example code

Although pipes are usually used with threads, this example simply writes data to a pipe and then reads it back later.

First, a `PipedReader` is constructed, then a `PipedWriter` that writes to that `PipedReader`. The attributes are written to the pipe as strings with a vertical bar as a delimiter and a new-line character placed at the end. The entire contents of the `PipedReader` are read as a `String` and displayed.

We'll show later, in Tokenizing on page 54, how to use a `StringTokenizer` to break up a string that is delimited into individual values.

Here is the example code:

```
import java.io.*;
import java.util.*;
class PipedExample
     {
     static BufferedReader system_in = new BufferedReader
            (new InputStreamReader(System.in));

     public static void main(String argv[])
           {
           PipedReader pr = new PipedReader();
           PipedWriter pw = null;
           try {
                 pw = new PipedWriter(pr);
                 }
           catch (IOException e)
                 {
           System.err.println(e);
                 }

           // Create it {
           // Read in three hotels
           for (int i = 0; i < 3; i++)
                 {
                 Hotel a_hotel = new Hotel();
                 a_hotel.input(system_in);
                 a_hotel.write_to_pw(pw);
                 }
           }

           // Print it
           {
           char [] buffer = new char[1000];
           int length = 0;
           try
                 {
                 length = pr.read(buffer);
                 }
           catch (IOException e)
                 {
                 System.err.println(e);
                 }
           String output =new String(buffer, 0, length);
             System.out.println("String is ");
           System.out.println(output);
           }

           }
     }

class Hotel
```

```
      {
      private String name;
      private int rooms;
      private String location;
      boolean input(BufferedReader in)
            {
            try
                  {
                  System.out.println("Name: ");
                  name = in.readLine();
                  System.out.println("Rooms: ");
                  String temp = in.readLine();
                  rooms = to_int(temp);
                  System.out.println("Location: ");
                  location = in.readLine();
                       }
            catch(IOException e)
                  {
                  System.err.println(e);
                  return false;
                  }
            return true;
            }
      boolean write_to_pw(PipedWriter pw)
            {
            try
                  {
                  pw.write(name);
                  Integer i = new Integer(rooms);
                  pw.write(i.toString());
                  pw.write(location);
                  pw.write('backslash n');
// red font indicates that an actual backslash n (carriage return character)
// should be inserted in the code.
                  }
            catch(IOException e)
                  {
                  System.err.println(e);
                  return false;
                  }
            return true;
                    }

      void debug_print()
            {
            System.out.println("Name :" + name +
                  ": Rooms : " + rooms + ": at :" + location+ ":");
            }
      static int to_int(String value)
            {
            int i = 0;
            try
                  {
                  i = Integer.parseInt(value);
                  }
            catch(NumberFormatException e)
                  {}
            return i;
            }
      }
```

---

# Sequences

`SequenceInputStream` combines input streams and reads each input stream until the end.

You can use `SequenceInputStream` to read two or three streams as if they were one stream. You can concatenate streams from various sources, such as two or more files. To construct a `SequenceInputStream`, you can specify either two streams or an Enumeration, which represents a set of input streams. A program such as "cat" in UNIX would use something like this.

`SequenceInputStream` reads from the first underlying input stream that it is given. When the first one is exhausted, it opens the next input stream and reads that one. When that one is exhausted, it reads the next, and so on.

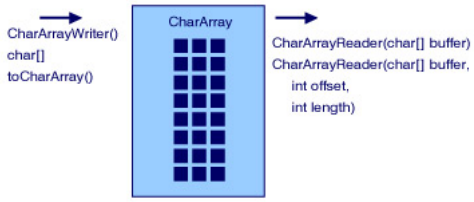
The user does not know when the input is transferred from one stream to the next stream; another byte is simply read. When all the input streams have been read, an EOF is received from the entire input stream.

---

# Char arrays

Now let's explore `Readers` and `Writers`. The read and write methods input and output a character. Similar to `ByteArrayOutputStream`, a `CharArrayWriter` writes characters to a char array. As a character is written to a `CharArrayWriter` object, it's added to an array of characters whose size is automatically incremented.

At any point in time, we can get the character array that we have filled up. The `toCharArray()` method returns an array of characters. A `CharArrayReader` uses a character array as a source. Typically, the array is one that has been created with a `CharArrayWriter` object. With an alternative constructor, you can specify not only the array, but where to start in the array (an offset) and how many bytes to read (a length) before you return an EOF character.




---

# String: Example code

`StringWriter` works like `CharArrayWriter`. An internal `StringBuffer` object is the destination of the characters written. Methods associated with the class are `getBuffer()`, which returns the `StringBuffer` itself, and `toString()`, which returns the current value of the string.

`StringReader` works like `CharArrayReader`. It uses a `String` object as the source of the characters it returns. When a `StringReader` is created, you must specify the `String` that it is read from.

In this example, instead of writing to a pipe, we are going to write to a `String`. A `StringWriter` object is constructed. After the output is complete, we obtain the contents with `toString()` and print it out. This works like the previous example with `PipedReader` and `PipedWriter`, except a `String` is used to contain the data.

Here is the example code:

```
import java.io.*;
import java.util.*;
class StringExample
     {
     static BufferedReader system_in = new BufferedReader
            (new InputStreamReader (System.in));

     public static void main(String argv[])
           {
           StringWriter sw = new StringWriter();

           // Create it
           {
           // Read in three hotels
           for (int i = 0; i < 3; i++)
                 {
                 Hotel a_hotel = new Hotel();
                 a_hotel.input(system_in);
                 a_hotel.write_to_string(sw);
                 }
           }

           // Print it
                 {
           String output = sw.toString();
           System.out.println("String is ");
           System.out.println(output);
           }

           }
     }

class Hotel
     {
     private String name;
     private int rooms;
     private String location;
     boolean input(BufferedReader in)
           {
           try
                 {
                 System.out.println("Name: ");
                 name = in.readLine();
                 System.out.println("Rooms: ");
                 String temp = in.readLine();
                 rooms = to_int(temp);
                 System.out.println("Location: ");
                 location = in.readLine();
           }
           catch(IOException e)
                 {
                 System.err.println(e);
                 return false;
                 }
           return true;
           }
     boolean write_to_string(StringWriter sw)
           {
           sw.write(name);
```

```
            Integer i = new Integer(rooms);
            sw.write(i.toString());
            sw.write(location);
            sw.write('backslash n');
 // red font indicates that an actual backslash n (carriage return character)
 // should be inserted in the code.
            return true;
            }

      void debug_print()
            {
            System.out.println("Name :" + na
           ": Rooms : " + rooms + ": at :" + location+ ":");
            }
      static int to_int(String value)
            {
            int i = 0;
            try
                  {
                  i = Integer.parseInt(value);
                  }
            catch(NumberFormatException e)
                  {}
            return i;
            }
      }
```

---

# InputStreamReader

`InputStreamReader` reads bytes from an `InputStream` and converts them to characters. An `InputStreamReader` uses the default character encoding for the bytes, which is usually ASCII. If the bytes that are being read are ASCII bytes, only a byte at a time is used to form a character.

If the bytes are not ASCII, such as Chinese or another language, you want conversion to Unicode as well. Specific encoding of the byte stream is necessary, and the `InputStreamReader` converts it to Unicode. With an alternate constructor, you can specify the encoding of the bytes on the `InputStream`.

---

# OutputStreamReader

`OutputStreamWriter` is similar to `InputStreamReader`. The output characters, which are in Unicode, are converted to the underlying format of the machine using an `OutputStreamWriter`. The converted characters are written to an `OutputStream`. The underlying default format is typically ASCII. However, you can state a specific encoding scheme with an alternate constructor.

# Section 5. Files

## Introduction

This section examines the `File` class, an important non-stream class that represents a file or directory name in a system-independent way. The `File` class provides methods for:

* Listing directories
* Querying file attributes
* Renaming and deleting files

---

## The File classes

The `File` class manipulates disk files and is used to construct `FileInputStreams` and `FileOutputStreams`. Some constructors for the `File` I/O classes take as a parameter an object of the `File` type. When we construct a `File` object, it represents that file on disk. When we call its methods, we manipulate the underlying disk file.

The methods for `File` objects are:

* Constructors
* Test methods
* Action methods
* List methods

---

## Constructors

Constructors allow Java code to specify the initial values of an object. So when you're using constructors, initialization becomes part of the object creation step. Constructors for the `File` class are:

```
*   File(String filename)
*   File(String pathname, String filename)
*   File(File directory, String filename)
```

---

## Test Methods

Public methods in the `File` class perform tests on the specified file. For example:

* The `exists()` method asks if the file actually exists.
* The `canRead()` method asks if the file is readable.
* The `canWrite()` method asks if the file can be written to.
* The `isFile()` method asks if it is a file (as opposed to a directory).

* The `isDirectory()` method asks if it is a directory.

These methods are all of the boolean type, so they return a true or false.

---

## Action methods

Public instance methods in the `File` class perform actions on the specified file. Let's take a look at them:

* The `renameTo()` method renames a file or directory.
* The `delete()` method deletes a file or directory.
* The `mkdir()` method creates a directory specified by a File object.
* The `mkdirs()` method creates all the directories and necessary parents in a File specification.

The return type of all these methods is boolean to indicate whether the action was successful.

---

## List methods

The `list()` method returns the names of all entries in a directory that are not rejected by an optional `FilenameFilter`. The `list()` method returns null if the `File` is a normal file, or returns the names in the directory. The `list()` method can take a `FilenameFilter` filter and return names in a directory satisfying the filter.

---

## FilenameFilter interface

The `FilenameFilter` interface is used to filter filenames. You simply create a class that implements the `FilenameFilter`. There are no standard `FilenameFilter` classes implemented by the Java language, but objects that implement this interface are used by the `FileDialog` class and the `list()` method in the `File` class.

The implemented `accept()` method determines if the filename in a directory should be included in a file list. It is passed the directory and a file name. The method returns true if the name should be included in the list.

---

## File class: Example code

This example shows a file being tested for existence and a listing of the C:\Windows directory. The listing is performed for all files and then for files matching a filter.

Here is the example code:

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

class FileClassExample
     {
     public static void main(String argv[])
           {
           File a_file = new File("test.txt");
           if (a_file.canRead())
                 System.out.println("Can read file");
           if (a_file.isFile())
                 System.out.println("Is a file");

           File a_directory = new File("C:backslash, backslashWindows");
// red font indicates that an actual backslash n (carriage return character)
// should be inserted in the code.
           if (a_directory.isDirectory())
                 {
                 System.out.println("Is a directory");
                 String names[] = a_directory.list();
                 for (int i = 0; i < names.length; i++)
                       {
                       System.out.println("Filename is " + names[i]);
                       }
                                    }
           System.out.println("Parent is " + a_directory.getParent());

           if (a_directory.isDirectory())
                 {
                 String names[] = a_directory.list(new MyFilter());
                 for (int i = 0; i < names.length; i++)
                       {
                       System.out.println("Filename is " + names[i]);
                       }
                 }

     }
     }

class MyFilter implements FilenameFilter
     {
     public boolean accept(File directory, String name)
           {
           if (name.charAt(0) = = 'A' || name.charAt(0) = = 'a')
           return true;
                      }
     }
```

---

# FileInputStream and FileOutputStream

You can open a file for input or output.

`FileInputStream` reads from a disk file. You can pass that constructor either the name of a file or a `File` object that represents the file. The `FileInputStream` object is a source of data.

`FileOutputStream` writes to a disk file. You can pass it a `File` object or a name. The `FileOutputStream` object is a sink for data.

For `FileInputStream` and `FileOutputStream`, you read and write bytes of data.

# File: Example code

This example works like the previous examples, except the output is to a file. We open the file and write the data as bytes. After closing the file, we open it for reading, read all the bytes in the file, and print them as a string.

Here is the example code:

```
import java.io.*;
import java.util.*;
class FileExample
     {
     static BufferedReader system_in = new BufferedReader
            (new InputStreamReader(System.in));

     public static void main(String argv[])
           {

           // Create it
           {
           try
                 {
                 FileOutputStream fos = new FileOutputStream("file.dat");
                 // Read in three hotels
                 for (int i = 0; i < 3; i++)
                       {
                       Hotel a_hotel = new Hotel();
                       a_hotel.input(system_in);
                       a_hotel.write_to_fos(fos);
                       }
                 fos.close();
                 }
           catch(IOException e)
            {
                 System.out.println(e);
                 }
           }

           // Now display it
           {
           byte [] buffer = null;
           File a_file = new File("file.dat");
           System.out.println("Length is " + a_file.length());
           System.out.println(" Can read " + a_file.canRead());
           try
                 {
                 FileInputStream fis = new FileInputStream(a_file);
                 int length = (int) a_file.length();
                 buffer = new byte[length];
                fis.read(buffer);
                fis.close();
                 }
                 catch(IOException e)
                 {
                 System.out.println(e);
                 }

           String s = new String(buffer);
           System.out.println("Buffer is " + s);
           }

           }
     }
```

```
class Hotel
     {
     private String name;
     private int rooms;
     private String location;
     boolean input(BufferedReader in)
           {
           try
                 {
                 System.out.println("Name: ");
                  name = in.readLine();
                 System.out.println("Rooms: ");
                 String temp = in.readLine();
                 rooms = to_int(temp);
                 System.out.println("Location: ");
                 location = in.readLine();
                 }
           catch(IOException e)
                 {
                 System.err.println(e);
                 return false;
                 }
           return true;
           }
     boolean write_to_fos(FileOutputStream fos)
           {
           try
                 {
                 fos.write(name.getBytes());
                 Integer i = new Integer(rooms);
                 fos.write(i.toString().getBytes());
                 fos.write(location.getBytes());
                 fos.write(' ');
                 }
                    catch (IOException e)
                 {
                 System.err.println(e);
                 return false;
                 }
           return true;
           }
           void debug_print()
           {
           System.out.println("Name :" + name +
                 ": Rooms : " + rooms + ": at :" + location+ ":");
           }
     static int to_int(String value)
           {
           int i = 0;
           try
                 {
                 i = Integer.parseInt(value);
                 }
           catch(NumberFormatException e)
                 {}
           return i;
           }
     }
```

---

# FileReader and FileWriter

`FileReader` is a convenience subclass of `InputStreamReader`. `FileReader` is useful when you want to read characters from a file. The constructors of `FileReader` assume a

default character encoding and buffer size. `FileReader` constructors use the functionality of `InputStreamReader` to convert bytes using the local encoding to the Unicode characters used by Java code.

If you want to read Unicode characters from a file that uses encoding other than the default, you must construct your own `InputStreamReader` on `FileInputStream`.

`FileWriter` is a convenience subclass of `OutputStreamWriter` for writing character files. Constructors for `FileWriter` also assume default encoding and buffer size. If you want to use encoding other than the default, you must create your own `OutputStreamWriter` on `FileOutputStream`.

# Section 6. Buffering

## Introduction

This section introduces buffering and covers the following topics:

* `BufferedInputStream` and `BufferedOutputStream`
* `BufferedReader` and `BufferedWriter`
* New-line characters
* Buffering input and output streams

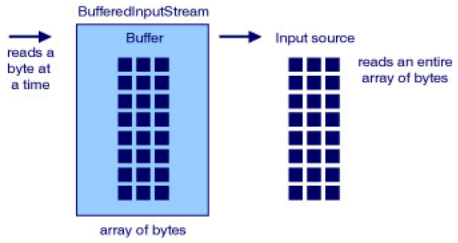
---

## What is buffering?

Reading and writing to a file, getting data one byte at a time, is slow and painstaking. One way to speed up the process is to put a wrapper around the file and put a buffer on it.

Our `BufferedInputStream` is going to put a buffer onto an `InputStream` that is specified in the constructor. The actual data source is what you pass it as an `InputStream`. The `BufferedInputStream` reads large chunks of data from the `InputStream`. Then you read individual bytes or small chunks of bytes from the `BufferedInputStream`. The default buffer size is 512 bytes, but there's a constructor that allows you to specify the buffer size if you want something different.

To improve your efficiency, you read from the object of `BufferedInputStream` instead of reading directly from the underlying `InputStream`. And you won't have to go back to the operating system to read individual bytes.



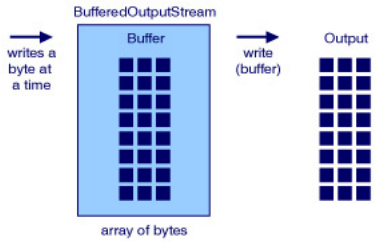

---

## BufferedOutputStream

`BufferedOutputStream` extends `FilterOutputStream`. When you apply it to an `OutputStream`, you have a buffered output stream. Instead of going to the operating system for every byte you write, you have the intermediary that provides a buffer and you write to that.

When that buffer is full, it is written all at once to the operating system. And it is written automatically if the buffer gets full, if the stream is full, or if the `flush()` method is used. The `flush()` method forces any output buffered by the stream to be written to its destination. So for a `BufferedOutputStream`, you have tell it:

* The output stream you are going to use
* The buffer size if you don't like the default




---

# BufferedReader and BufferedWriter

A `BufferedReader` and a `BufferedWriter` act like `BufferedOutputStream` and `BufferedInputStream`, except they deal with reading and writing characters. For a `BufferedReader`, you specify an underlying `Reader` and optionally a buffer size. For a `BufferedWriter`, you specify an underlying `Writer` and optionally a buffer size.

`BufferedReader` has one additional method, called `readLine()`, which allows us to simply read an entire line of characters from the underlying `Reader`.

---

# BufferedReader: Example code

If you've been using Java 1.0, the `readLine()` method was actually part of `DataInputStream`. Now you should be using a `BufferedReader` for `readLine()`, even though you can still do that with a `DataInputStream`.

A `DataInputStream` reads bytes but you are really reading characters when you read lines, so using `readLine()` and the `BufferedReader` is the preferred way.

Here is the example code:

```
import java.io.*;
import java.util.*;
class TextReaderWriterExample
     {
     static BufferedReader system_in = new BufferedReader
                 (new InputStreamReader(System.in));

     public static void main(String argv[])
           {
           // Create it
           {
           try
                 {
                 FileOutputStream fos = new FileOutputStream("text.dat");
                 PrintWriter pw = new PrintWriter(fos);

                 for (int i = 0; i < 3; i++)
                       {
                       Hotel a_hotel = new Hotel();
```

```
                        a_hotel.input(system_in);
                        a_hotel.write_to_pw(pw);
                        }
                  pw.close();
                  }
            catch(IOException e)
               {
                  System.err.println(e);
                  }
            }

            // Now read it
            {
            try
                  {
                    FileReader fr = new FileReader("text.dat");
                  BufferedReader br = new BufferedReader(fr);
                  Hotel a_hotel = new Hotel();
                  while (a_hotel.read_from_br(br))
                        {
                        a_hotel.debug_print();
                        }
                  br.close();
                  }
            catch(IOException e)
                  {
                  System.err.println(e);
                  }
            }

            }
             }

class Hotel
     {
     private String name;
     private int rooms;
     private String location;
     boolean input(BufferedReader in)
            {
            try
                  {
                  System.out.println("Name: ");
                  name = in.readLine();
                  System.out.println("Rooms: ");
                  String temp = in.readLine();
                  rooms = to_int(temp);
                  System.out.println("Location: ");
                  location = in.readLine();
                  }
            catch(IOException e)
                  {
                  System.err.println(e);
                  return false;
                  }
            return true;
            }
             boolean write_to_pw(PrintWriter pw)
            {
            pw.print(name);
            pw.print('|');
            pw.print(rooms);
            pw.print('|');
            pw.print(location);
            pw.println();
            return true;
            }
     boolean read_from_br(BufferedReader br)
            {
```

```
        try
              {
              String line = br.readLine();
              if (line == null)
                    return false;
              StringTokenizer st = new StringTokenizer(line, "|");
              int count = st.countTokens();
              if (count < 3)
                    return false;
              name = st.nextToken();
              String temp = st.nextToken();
              rooms = to_int(temp);
              location = st.nextToken();
              }
              catch(IOException e)
              {
              System.err.println(e);
              return false;
              }
        return true;
        }
  void debug_print()
        {
        System.out.println("Name :" + name +
              ": Rooms : " + rooms + ": at :" + location+ ":");
        }
  static int to_int(String value)
        {
        int i = 0;
        try
              {
              i = Integer.parseInt(value);
              }
        catch(NumberFormatException e)
              {}
        return i;
        }
```

---

# BufferedWriter

As we mentioned, a `BufferedWriter` allows us to write to a buffer. It applies buffering to a character output stream, improving output efficiency by combining many small write requests into a single larger request.

This class has an interesting additional method that makes Java code portable. Most of the time, a new line is represented by a `\n`, but it may not be in all operating systems. Because of this, the Java language adds a method called `newLine()`.

Instead of outputting the character `\n`, you call `newLine()` and that outputs the appropriate new-line character for the particular operating system that you are running on. In most cases, this will be `\n`. The new-line character that is written is the one returned by passing `line.separator` to the `getProperty()` method in the `System` class.

---

# Expressing a new line

So you may ask, how does the Java language know how to express a new line for a particular operating system?

The Java `newLine()` method asks the system, "What is your new-line character?" In the Java language, this is called a system property. There is a `System` class with properties. When you say, "What's your new-line character?" by passing `line.separator` to the `getProperty()` method in the `System` class, you get an answer back. Depending on the platform, the new-line character can be a new-line character, a carriage-return character, or both.

The `newLine()` method, which is part of `BufferedWriter`, outputs the platform-dependent line separator to the stream by using such a call.

---

## When to use BufferedWriter

**PrintWriter() API:**

```
PrintWriter out = new

PrintWriter (new BufferedWriter
    (new FileWriter ("file.out"));
```

You typically use a `BufferedWriter` for your output. The only time you don't want to have a buffered output is if you are writing out a prompt for the user. The prompt would not come up until the buffer was full, so for those sorts of things you do not want to use buffering.

---

# Buffering input and output streams: Example code

Here's an example of some code that shows the efficiency of buffering. Typically the `read()` method on `InputStreamReader` or `FileReader` performs a read from the underlying stream.

It might be more efficient to wrap a `BufferedReader` around these `Readers`. With buffering, the conversion from byte to character is done with fewer method invocations than if the `InputStreamReader read()` method is called directly.

Likewise, the `write()` method on an `OutputStreamWriter` or `FileWriter` performs a write to the underlying stream. It can be more efficient to wrap a `BufferedWriter` around these `Writers`.

Here is the example code:

```
BufferedReader in=
  newBufferedReader(newinputStreamReader(System.in));
Writer out=
  newBufferedWriter(new FileWriter("file.out");
```

# Section 7. Filtering

## Introduction

This section introduces filtering and covers the following topics:

* `PushbackInputStream`
* `PushbackReader`
* `LineNumberReader`
* `PrintWriter`

---

## What is filtering?

Use filtering to read or write a subset of data from a much larger input or output stream. The filtering can be independent of the data format (for example, you need to count the number of items in the list) or can be directly related to the data format (for example, you need to get all data in a certain row of a table). You attach a filter stream to an input or output stream to filter the data.

`FilterInputStream` and `FilterOutputStream` filter input and output bytes. When a `FilterInputStream` is created, an `InputStream` is specified for it to filter. Similarly, you specify an `OutputStream` to be filtered when you create a `FilterOutputStream`. They wrap around existing `InputStream` or `OutputStream` to provide buffering, checksumming, and so on. (We covered buffering in Buffering on page 25 .)

For character I/O, we have the abstract classes `FilterWriter` and `FilterReader`. `FilterReader` acts as a superclass for character input streams that read data from some other character input stream, filter it, and then return the filtered data when its own `read()` methods are called. `FilterWriter` is for character output streams that filter data written to them before writing it to some other character output stream.

---

**Pushback APIs:**

```
PushbackInputStream(InputStream is)
void unread(int byte)
void unread(byte [] array)

PushbackReader(Reader in)
void unread(int character)
void unread(char [] buffer)
```

# Pushback

You use `PushbackInputStream` to implement a one-byte pushback buffer or a pushback buffer of a specified length. It can be used to parse data. When would you use this?

Sometimes when you read a byte from an `InputStream`, it signifies the start of a new block of data (for example, a new token). You might say, "I have read this byte, but I really cannot deal with it at this time. I want to push it back on the `InputStream`. When I come back later to read the `InputStream`, I want to get that same byte again."

You push the byte back onto the stream with the `unread()` method. When you read it again, you will read the data you unread before. You can `unread()` a single byte, an array of bytes, or an array of bytes with an offset and a length.

If you are reading characters, you can push back a character instead of a byte. You use `PushbackReader`, which works just like the `PushbackInputStream`.

Use the `unread()` methods to unread a single character, an array of characters, or an array of characters with an offset and a length.

**LineNumberReader APIs:**

```
LineNumberReader(Reader r)

LineNumberReader(Reader r,
  int buffer_size)

String readLine()

int getLineNumber()

void setLineNumber(int line_num)
```

## LineNumberReader versus LineNumberInputStream

A `LineNumberReader` is a character input stream that keeps track of the number of lines of text that have been read from it. A line is considered terminated by a new line (`\n`), a carriage return ), or a carriage return followed by a linefeed. The `readLine()` method returns all the characters in the line without the terminator. The `getLineNumber()` method returns the number of lines that have been read.

If, for some reason, you want to reset the line number, you can do that with `setLineNumber()` and start the line count again from that point.

`LineNumberInputStream` also keeps track of the number of lines of data that have been read, but this class was deprecated in Java 1.1 because it reads bytes rather than characters. `LineNumberReader` was introduced, which reads characters.

---

## PrintStream versus PrintWriter

`PrintStream` is used to output Java data as text. It implements a number of methods for displaying textual representation of Java primitive data types.

You've probably used a `PrintStream` object since your earliest Java programming days. `System.out` is a `PrintStream` object. Probably everyone has used that for debugging! That being said, `PrintStream` has been superseded by `PrintWriter` in Java 1.1. The constructors of this class have been deprecated, but the class itself has not because it is still used by the `System.out` and `System.err` standard output streams.

The methods in `PrintStream` and `PrintWriter` are very similar. `PrintWriter` does not have methods for writing raw bytes, which `PrintStream` has.

---

## Flushing

Both the `PrintStream` and `PrintWriter` classes employ flushing, which moves data from the internal buffers to the output stream. Flushing is employed in two ways:

* Automatic flushing, which says when you call `println` you get flushing

* Without automatic flushing, which says flushing occurs only when the `flush()` method is called

With the `PrintStream` class, if a new-line character was written, the output is flushed. With the `PrintWriter` class, if you enable automatic flushing, the output is flushed only when a `println()` method is invoked. The `println()` methods of `PrintWriter` use the `System` property `line.separator` instead of the new-line (`\n`) character that `PrintStream` uses.

---

**PrintWriter() APIs:**

```
PrinterWriter (Writer w)

PrinterWriter (Writer w,
  Boolean autoflush)

PrinterWriter (OutputStream os)

PrinterWriter (OutputStream os,
  Boolean autoflush)
```

## PrintWriter process

Let's look at the whole process.

You can give the `PrintWriter` constructor a `Writer` or you can give it an `OutputStream`. With the latter, an `OutputStreamWriter` is transparently created, which performs the conversion for characters to bytes using the default character encoding.

You can choose one of the two flushing options: with autoflush or without. You can print integers, strings, and characters without a new line with the `print()` methods and with a new line with the `println()` methods.

# Section 8. Checksumming

## Introduction

This section explores checksumming. `Checksum` is an interface that several different classes implement. Let's take a look at what this section covers:

* What is a checksum and how does it work?
* What is a message digest and how does it work?
* Checksum versus message digest

---

**Checksum methods:**

```
long getValue()

void reset()

void update(byte [] array,
  in offset, int length)
```

## What is checksumming?

What's a checksum? The `Checksum` interface defines the methods required to compute a checksum on a set of data. The computation of a checksum is such that if a single bit or two bits in the data are changed, the value for the checksum changes.

A checksum is computed based on the bytes of data supplied to the `update()` method. The `update()` method updates the current checksum with an array of bytes by adding it to an ultimate value.

Additionally, the current value of the checksum can be obtained at any time with the `getValue()` method. Use the `reset()` method to reset the default value to its initial value, usually 0.

There are two types of checksums. The most common is the CRC32, that's Cycle Redundancy Check. Another type of checksum is the Adler32, used for computing Adler32 checksum. The Adler32 has a faster computation. It's nearly as reliable as the CRC32.

---

## CheckedInputStream and CheckedOutputStream

Let's create a `Checksum` object and use it to filter an input or output stream. When writing a byte to the `CheckedOutputStream`, the checksum is automatically computed.

`CheckedOutputStream` calls the `update()` method in the `Checksum` object. Then the byte

goes out to its ultimate destination. At any point in time, you can ask the `Checksum` object for the current checksum. You can write all your bytes out and add that checksum to what you have written out.

* `CheckedInputStream` implements `FilterInputStream` and maintains a checksum on every read.
* `CheckedOutputStream` implements `FilterOutputStream` and maintains a checksum on every write.

---

## Adding checksum

Use a checksum to make sure that data was transmitted properly. Construct a `CheckedOutputStream` using an `OutputStream` and an object of the `Checksum` type. CRC32 and Adler32 are the two types of checksums you can choose from or you can create your own. At periodic times, say every 512 bytes, retrieve the current value of the checksum and send those four bytes over the stream.

On the receiving end, you construct a `CheckedInputStream` using an `InputStream` and an object of the same type you used for the `CheckedOutputStream`. After reading 512 bytes, retrieve the current value of the checksum and compare it to the next four bytes read from the stream.

---

## CheckedOutputStream: Example code

Let's look at some sample code. We are going to open a `FileOutputStream`, `Temp1.tmp`. An object of the CRC32 type is constructed, and that is used to create a `CheckedOutputStream`. When writing to the file output a byte at a time, the checksum is computed. At the end, the `Checksum` object is then asked the current value of the checksum.

Checksumming only gives you a 32-bit value. If a single bit changes, you will notice it, but if lots of bits change, you could possibly get the same checksum.

Here is the example code:

```
FileOutputputStream os =
  new FileOutputStream("Temp1.tmp");
CRC32 crc32 = new CRC32();
CheckedOutputStream cos =
   new CheckedOutputStream(os, crc32);
cos.write(1);
cos.write(2);
long crc = crc32.getValue();
```

---

**Digesting methods:**

```
static MessageDigest
  getInstance(String algorithm)

void update (byte [] array)

byte [] digest()
```

## Digesting

Let's create something larger than a `Checksum`. A larger set of bits that represents a sequence of bytes is a message digest. A message digest works like a checksum. It is a one-way hash that takes a variable amount of data and creates a fixed-length hash value. This is called a digital fingerprint.

If somebody makes changes in the original sequence of things and manipulates the contents, the message digest is not going to look the same. This is how we implement security. There is a possibility, but it is very small, that if the message were changed, you would get the same message digest.

A sequence of bytes is transformed into a digest. In the `MessageDigest` class, we have a static function called `getInstance`. `getInstance` is given the name of the `MessageDigest` that we want to be using. There are two strings that you can pass it: SHA1 and MD5. Their details are in algorithmic books.

Additionally, the `update()` methods update the digest with the byte or array of bytes. The `digest()` method computes and returns the value of digest. Digest is then reset.

---

## DigestInputStream and DigestOutputStream

We have two equivalent methods for digest: the `DigestInputStream` and `DigestOutputStream`. Give the `DigestInputStream` an input stream and the `MessageDigest` that will be doing its calculations. You can turn digesting on or off for a stream. You may want to input or output some data, but not calculate it as part of the `MessageDigest`.

The `read()` and `write()` methods update the `MessageDigest` in `DigestInputStream` and `DigestOutputStream` respectively.

---

## MessageDigest: Example code

The first line creates a `FileOutputStream` going to the file `temp.tmp`. Create a

`MessageDigest` object. As you can see in the code `md = MessageDigest.getInstance()`, you're passing it the name of the message digesting algorithm that you want to use. It returns back to you an object of a `MessageDigest` type. If the string that you passed it is not the name of a digest, it throws a `NoSuchAlgorithmException`.

After you have the `MessageDigest`, use that to construct a `DigestOutputStream`. Bytes are written out to the underlying `OutputStream` and added to the digest. You can get `MessageDigest` by calling `md.digest()`. It returns the digest or an array of bytes. Now you can store that away or send it along.

Here is the example code:

```
FileOutputStream os = new FileOutputStream("Temp.tmp");
MessageDigest md = null;
try
     {
     md = MessageDigest.getInstance("SHA");
     }
catch(NoSuchAlgorithmException e)
     {
     System.out.println(e);
     }

DigestOutputStream dos = new DigestOutputStream(os, md);
dos.write(1);
dos.write(2);
byte [] digest = md.digest();
```

---

## Checksum versus MessageDigest

Remember, a checksum is small. A checksum is typically used to verify the integrity of data over a noisy transmission line or to verify whether a file contains the same data. A message digest, on the other hand, is much larger. It is used more for security to insure that a message has not been tampered with.

# Section 9. Inflating and deflating

## Introduction

This section explores inflating and deflating in Java code. These terms have the same meaning as compressing and uncompressing. Some of the key points in this section are:

* `DeflaterInputStream` and `DeflaterOutputStream`
* `ZipInputStream` and `ZipOutputStream`
* `ZipFile`

---

## Deflater

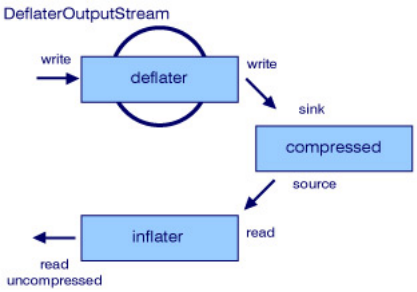
Let's move into deflating territory. Deflating writes a compressed representation of bytes to an output stream. Inflating takes those compressed bytes from an input stream, reads them in, and automatically uncompresses them. This process is similar to zipping and unzipping on the fly.

The algorithm of a `Deflater` is applied to the bytes output to a stream, and the corresponding algorithm of an `Inflater` is applied to the bytes input from a stream. The default `Deflater` and `Inflater` use the ZLIB compression library for compression and decompression. An alternate constructor allows for GZIP and PKZIP compression.

---

## DeflaterOutputStream

Let's journey into the output stream. You can simply create a `DeflaterOutputStream` with an output sink. As we write to the `DeflaterOutputStream`, the bytes are compressed using the `Deflater` algorithm and only the compressed bytes are sent to the underlying output stream. The default constructor for `DeflaterOutputStream` uses ZLIB compression with a default-sized buffer. With the other constructors, you can specify a different `Deflater` and a different-sized buffer.

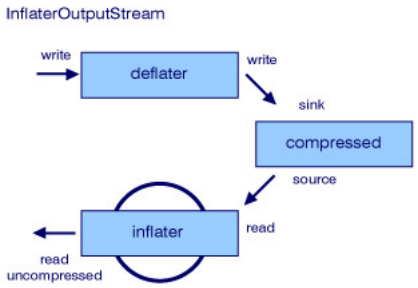



---

## InflaterInputStream

On the receiving side, an `InflaterInputStream` is implemented. When we construct it, we specify an input source. It uncompresses the bytes from that source as you go along. Once again, you can specify your own `Inflater` if you use a different compression scheme.

If you decide to use your own `Deflater` or `Inflater` instead of the default, then it is up to you to make sure the `Inflater` and `Deflater` match and understand each other.




---

## GZIP input and output stream

A `GZIPOutputStream` outputs the compressed bytes to the underlying stream in GZIP format. It is derived from the `DeflaterOutputStream`. The constructor is supplied an `OutputStream`. Correspondingly, a `GZIPInputStream` inputs from a source that is in GZIP format. It is derived from `InflaterInputStream`, and the constructor is supplied an `InputStream`.

Use `GZIPInputStream` and `GZIPOutputStream` on a single stream, which could be a file. However, it can be more efficient than a zip stream because it does not require creating a directory of zipped files.

---

## ZipFile: Example code

Let's look into the `ZipFile` and `ZipEntry` classes. These are not actually in the I/O hierarchy but they are good to know about because lots of people have zip files.

The `ZipFile` class allows us to read a zip file. A zip file consists of two parts: a list of entries of what's been zipped and the data itself. You read the set of entries using `entries()` and pick out a particular one to unzip. A `ZipEntry` object represents the file you have selected to uncompress. For that particular `ZipEntry`, you can obtain an `InputStream` using `getInputStream()` which, when read, returns the uncompressed data. The Java library is kept in a jar file, which is in zip file format. You can use a `ZipFile` to read a jar file if you want.

Here is the example code:

```
import java.util.zip.*;
import java.util.*;
import java.io.*;
```

```
class ZipFileExample
     {
     public static void main(String argv[])
           {
           try
                 {
                 ZipFile zf = new ZipFile("test.zip");
                 // Enumerate the entries
                 Enumeration zip_entries = zf.entries();
                 String last_name = null;
                 while(zip_entries.hasMoreElements())
                       {
                       ZipEntry ze = (ZipEntry) zip_entries.nextElement();
                       System.out.println("Entry " + ze.getName());
                       last_name = ze.getName();
                       }

                 // Lets unpack the last one as an example
                 // Its name is in last_name
                 ZipEntry ze = zf.getEntry(last_name);
                 if (ze == null)
                       System.out.println("Cannot find entry");
                 else
                       {
                       BufferedReader r =
                             new BufferedReader(new InputStreamReader(
                             zf.getInputStream(ze)));
                       long size = ze.getSize();
                       if (size > 0)
                             {
                               System.out.println("File is " + size);
                             String line = r.readLine();
                             if (line != null)
                                   System.out.println("First line is " + line);
                             }
                       r.close();
                       }
                 }
           catch(IOException e)
                 {
                 System.out.println(e);
                 }
           }
     }
```

**ZipInputStream methods:**

`getNextEntry()`

`closeEntry()`

`read()`

## ZipInputStream

`ZipInputStream` is our next stop. You can read a zip file in a stream using this class. You get the first entry in the directory of what has been zipped. You can then read the data for the first entry, which is automatically unzipped. Then you get the next entry and unzip it, and so forth. `ZipInputStream` extends `InflaterInputStream`. The methods include getting the next entry from the table and closing off the entry. Closing the entry allows for the next entry to be read without having to finish reading the data for the current entry. The `read()` method in `ZIPInputStream` will return `-1` when the last of the entries is read.

---

# ZipInputStream: Example code

The code below is an example of unzipping a file. You are going to open `test.zip`, which is our zip file. That file is the source of a `ZipInputStream`. So `zis` is pointing to `test.zip`. The `getNextEntry()` method initially returns the first entry from the zip file table. A reference to that entry is now available.

If the entry is null, it means that end-of-file has been reached. Reading from the zip file from this point, data that is read corresponds to that entry and is automatically uncompressed. In this case, a `DataInputStream` has been wrappered and used to read it. The `dis` object is just going to read from the corresponding place in the zip file and close it. The next entry will open the file and start reading from the zip file again, reading the next entry.

Here is the example code:

```
import java.util.zip.*;
import java.io.*;
class ZipInputStreamExample
     {
     public static void main(String argv[])
           {
           try
                 {
                 FileInputStream fis = new FileInputStream("test.zip");
                 ZipInputStream zis = new ZipInputStream(fis);
                 ZipEntry ze;
                 // Get the first entry
                 ze = zis.getNextEntry();
                 if (ze == null)
                       System.out.println("End entries");
                 else
                       {
                 // Use it as stream or reader
                       DataInputStream dis = new DataInputStream(zis);
                       int i = dis.readInt();
                             System.out.println("File contains " + i);
                       zis.closeEntry();
```

```
                }
            zis.close();
            }
        catch(Exception e)
            {
            System.out.println(e);
            }
        }
```

---

**ZipOutputStream methods:**

`putNextEntry()`

`closeEntry()`

`write()`

## ZipOutputStream

A `ZipInputStream` reads a zip-formatted stream; a `ZipOutputStream` writes zip-formatted streams. This class allows us to create a zip file. `ZipOutputStream` extends `DeflaterOutputStream`. One method in `ZipOutputStream` is `putNextEntry()`, which puts the next entry into the list of compressed files. You need to create a zip entry for each file you are compressing.

The `putNextEntry()` method writes an entry and positions the stream for data output. You can start writing the next set of compressed data. Similar to the `close()` methods in other classes, the `closeEntry()` method closes a finished entry; you are ready to create the next one.

---

# ZipOutputStream: Example code

In the code below, an output file is constructed called `test.zip`. A `ZipOutputStream` is then constructed using that file. We want to compresss the file `test.txt` onto the zip file, so we construct a `ZipEntry` using that name.

`test.txt` is going to show up in the directory for the zip file. At this point, anything written to the `ZipOutputStream` is now data for that entry and is written in compressed form.

A `DataOutputStream` is wrappered around `zos`. Then we write an integer, and the entry is closed. At this point, another `ZipEntry` could now be created and written out. You could use `WinZip` to decompress this file or use the preceding example to read the file.

Here is the example code:

```
import java.util.zip.*;
import java.io.*;
class ZipOutputStreamExample
     {
     public static void main(String argv[])
```

```
        {
        try
              {
              FileOutputStream fos = new FileOutputStream("test.zip");
              ZipOutputStream zos = new ZipOutputStream(fos);
              ZipEntry ze = new ZipEntry("test.txt");
              zos.putNextEntry(ze);
              // Use it as output
              DataOutputStream dos = new DataOutputStream(zos);
              dos.writeInt (1);
              zos.closeEntry();
              // Put another entry or close it
              zos.close();
              }
        catch(Exception e)
              {
              System.out.println(e);
              }
        }
}
```

# Section 10. Data I/O

## Introduction

This section introduces data input and output:

* `DataInputStream`
* `DataOutputStream`

---

## What is Data I/O?

Several classes implement the `DataInput` and `DataOutput` interfaces. Let's talk about how they work.

The primitive data types, such as `int`s, may be represented differently on the various platforms that Java code runs on. For example, an `int` may be represented with the most significant byte first (called Big Endian), as on an IBM mainframe, or with the least significant byte first (called Little Endian), as on an IBM PC. If you wrote out an `int` in binary form with a C program on one platform and tried to read it in on the other platform, the values would not agree.

That's where `DataInput` and `DataOutput` come in. Now you can take a primitive, like an `int`, and write it out. It is written in a platform-independent form. You can read that `int` back in on any other Java virtual machine and, regardless of where it was produced, it will be turned back into an `int` of the proper value.

All the primitives, such as `double`s, `int`s, `short`s, `long`s, and so forth, can be written in a system platform-independent manner using the methods in `DataOutput`. Likewise, they can be read using the methods in `DataInput`.

---

**Unicode methods:**

```
void writeUTF(String s)

String readUTF()
```

## Unicode Text Format

A `String` object is written in Unicode Text Format or UTF for short. Strings are composed of two-byte Unicode characters. UTF is a method for compressing the most common Unicode values. If the `String` contains just ASCII characters, the values between 1 and 127 are written as a single byte. Values 128 - 2047 are written as two bytes. For some uncommon Unicode values, three bytes are used to represent their value. This expansion of some values is much rarer than the compression of ASCII values, so using UTF to represent a `String` is a net gain.

---

# DataInputStream and DataOutputStream: Example code

Two classes, `DataInputStream` and `DataOutputStream`, implement `DataInput` and `DataOutput`. Their methods include implementations for `readInt()`, `writeInt()`, and the other methods in the interfaces. You wrapper a `DataInputStream` around an . When `readInt()` is called, four bytes are read from the underlying stream and the resulting `int` is returned. Correspondingly, you wrapper a `DataOutputStream` around an OutputStream.

The code example below shows a `DataOutputStream` that uses a `FileOutputStream` as its underlying stream. The file is opened using a `FileInputStream`, which has a `DataInputStream` wrappered around it. The `data.dat` could be written on an IBM mainframe and read back on an IBM PC.

Here is the example code:

```
import java.io.*;
class DataInputOutputExample
     {

     static BufferedReader system_in = new BufferedReader
            (new InputStreamReader(System.in));

     public static void main(String argv[])
           {

           // Create it
           {
           try
                 {
                 FileOutputStream fos = new FileOutputStream("data.dat");
                 DataOutputStream dos = new DataOutputStream(fos);
                 // Read in three hotels
                 for (int i = 0; i < 3; i++)
                       {
                       Hotel a_hotel = new Hotel();
                       a_hotel.input(system_in);
                       a_hotel.write_to_dos(dos);
```

```
                        }
                  dos.close();
                              }
            catch(IOException e)
                  {
                  System.err.println(e);
                  }
            }

            // Now read it
            {
            try
                  {
                  FileInputStream fis = new FileInputStream("data.dat");
                  DataInputStream dis = new DataInputStream(fis);
                  Hotel a_hotel = new Hotel();
                  while (a_hotel.read_from_dis(dis))
                        {
                        a_hotel.debug_print();
                        }
                  dis.close();
                  }
            catch(IOException e)
                  {
                  System.err.println(e);
                  }
                         }

            }
      }

 class Hotel
      {
      private String name;
      private int rooms;
      private String location;
      boolean input(BufferedReader in)
            {
            try
                  {
                  System.out.println("Name: ");
                  name = in.readLine();
                  System.out.println("Rooms: ");
                  String temp = in.readLine();
                  rooms = to_int(temp);
                  System.out.println("Location: ");
                  location = in.readLine();
                  }
            catch(IOException e)
                  {
                  System.err.println(e);
                              return false;
                  }
            return true;
            }
      boolean write_to_dos(DataOutputStream dos)
            {
            try
                  {
                  dos.writeUTF(name);
                  dos.writeInt(rooms);
                  dos.writeUTF(location);
                  }
            catch(IOException e)
                  {
                  System.err.println(e);
                  return false;
                  }
            return true;
```

```
        }
    boolean read_from_dis(DataInputStream dis)
        {
        try
            {
            name = dis.readUTF();
               rooms = dis.readInt();
            location = dis.readUTF();
            }
        catch(EOFException e)
            {
            return false;
            }
        catch(IOException e)
            {
            System.err.println(e);
            return false;
            }
        return true;
        }
    void debug_print()
        {
        System.out.println("Name :" + name +
            ": Rooms : " + rooms + ": at :" + location+ ":");
        }
    static int to_int(String value)
        {
        int i = 0;
        try
            {
            i = Integer.parseInt(value);
            }
        catch(NumberFormatException e)
            {}
        return i;
        }

    }
```

---

# RandomAccessFile: Example code

`RandomAccessFile` implements both `DataInput` and `DataOutput`. You construct `RandomAccessFile` with a file name. Then you can use both read and write methods, as `readInt()` and `writeInt()`, on a `RandomAccessFile`.

Random access means you can position the next read or write to occur at a particular byte position within the file. You can obtain the current position with `getFilePointer()`. Later on, you can go back to that same position by passing it to `seek()`.

The example code below shows writing three `ints` to a file. The file position before the second write is stored in the pointer. After writing the next two `ints`, the position is restored to that pointer. The `readInt()` returns the value 15.

Here is the example code:

```
import java.io.*;

class RandomAccessFileExample
    {
```

```
public static void main(String argv[])
      {
      try
            {
            RandomAccessFile raf =
                  new RandomAccessFile("random.dat", "rw");
            raf.writeInt(12);
            long pointer = raf.getFilePointer();
            raf.writeInt(15);
            raf.writeInt(16);
            // Now read back the 2nd one
            raf.seek(pointer);
            int i = raf.readInt();
            System.out.println("Read " + i);
            }
      catch (IOException e)
            {
            System.out.println(e);
            }
         }
      }
```

# Section 11. Object serialization

## Introduction

This section introduces object serialization and covers the following topics:

* What is object serialization and how does it work?
* Features of a serializable class
* Externalizable

---

## What is object serialization?

Object serialization lets you take all the data attributes, write them out as an object, and read them back in as an object. Object serialization is quite useful when you need to use object persistence. GUI builders of JavaBeans use serialization to store the attributes of a JavaBean so that it can be accessed or modified by others.

You could actually use `DataOuputStreams` and `DataInputStreams` to write each attribute out individually, and then you could read them back in on the other end. But you want to deal with the entire object, not its individual attributes. You want to be able to simply store away an object or send it over a stream of objects. Object serialization takes the data members of an object and stores them away or retrieves them, or sends them over a stream.

You have the `ObjectInput` interface, which extends the `DataInput` interface, and `ObjectOutput` interface, which extends `DataOutput`, so you are still going to have the methods `readInt()` and `writeInt()` and so forth. `ObjectInputStream`, which implements `ObjectInput`, is going to read what `ObjectOutputStream` produces.

---

## How object serialization works

If you want to use the `ObjectInput` and `ObjectOutput` interface, the class must be serializable. How is a class serializable?

The serializable characteristic is assigned when a class is first defined. Your class must implement the `Serializable` interface. This marker interface says to the Java virtual machine that you want to allow this class to be serializable. You don't have to add any additional methods or anything.

There are a couple of other features of a serializable class. First, it has to have a zero parameter constructor. When you read the object, it needs to be able to construct and allocate memory for an object, and it is going to fill in that memory from what it has read from the serial stream.

The static fields, or class attributes, are not saved because they are not part of an object. If

you do not want a data attribute to be serialized, you can make it transient. That would save on the amount of storage or transmission required to transmit an object.

---

# Using ObjectOutputStream: Example code

Let's see how this works.

A `FileObjectStream` is created using `object.dat`, which is going to be used to store away objects.

You create `ObjectOutputStream` using that file. Any other `OutputStream`, such as that returned by `getOutputStream()` for a socket, could have been used. And now you have `YourClass`. This is whatever class you have made up. You create an object of that class. If you want to store that away on the `ObjectOutputStream`, you simply call `writeObject()` and pass it the object. That's it!

The class could have 20 attributes in it or 50 attributes; it doesn't matter. All of those attributes are automatically stored away. The class information is stored automatically.

The Java code knows what's in a class and knows the attributes in a class. In fact, the first time the code stores such an object, it stores a couple of markers about what the class is. Calling `writeObject()` is all that is necessary to store an object.

Here is the example code:

```
FileOutputSream fos = new FileOutputStream("object.dat");
ObjectOutputStream oos = new ObjectOutputStream(fos);
YourClass yc = new YourClass();
oos.writeObject(yc);
```

---

# Reading an object from a file: Example code

Now let's read that object from the file. Create a `FileInputStream` using the file `object.dat` that was written in the previous panel. That file is used as the source for an `ObjectInputStream`. You want to read the object, so we call `readObject()` and the object is read off the file. The attributes of the `YourClass` object are filled in with the data we have stored away.

The `readObject()` method returns a reference to an `Object` class object. You wrote out a `YourClass` object, so you have to cast the reference to `YourClass`. You have to keep track of the order in which you wrote things out. If you stored away a `YourClass` object, and then you stored away an `AnotherClass` object, you have to read them back in the same sequence.

Typically, you would write things out in a sequence, and then you would read them back in the same sequence. However, there are some alternative approaches, discussed in the next

panel.

Here is the example code:

```
FileInputStream fis = new FileInputStream ("object.dat");
ObjectInputStream ois = new ObjectInputStream (fis);
YourClass yc;
yc = (YourClass) ois.readObject();
```

---

## Alternative ways to read back: Example code

If you do not know the order in which the objects were stored, you can test for the class of the object that was returned by `readObject()` using `instanceof`, or you can use the `getClass()` method to determine the class of the object. Then you can assign the returned value to the appropriate object using a cast. You can do more elaborate reading and writing as you read back an object, test what it is, and then do an appropriate cast.

That's all you have to do to store away an object in a file. Now, you want to transmit it over a socket. All the values in that object are transmitted over a socket if you used the appropriate input and output streams from the socket object. Remote method invocation (RMI) passes objects using object serialization over a socket.

Here is the example code:

```
FileInputStream fis = FileinputStream("object.dat");
ObjectInputStream ois = new ObjectInputStream(fis);
Object object;
object = ois.readObject();
if (object instanceof YourClass)
     {
     YourClass yc = object;
     // Perform operations with yc;
     }
else if (object instance of AnotherClass)
     {
     AnotherClass ac = object;
     // Perform operations with ac
     }
```

---

## Add serializable: Example code

We said before that if the base class implements `Serializable`, all you have to do is mark your derived class `Serializable`. But what if your base class doesn't implement it? You can still implement `Serializable` in your derived class.

If the base class does not implement `Serializable` but you want to use serialization, you simply have to get all the attributes out of the base class and store them away. Your additional attributes are stored automatically, but the base class attributes are not. You have to use `get()` methods, and if the base class does not have gets and sets for its attributes, then you won't be able to serialize the base class. You need to define your own `readObject()` and

`writeObject()` methods, which we describe next.

Here is the example code;

```
class ABaseClass
     {
     private int value;
     int getValue()
           {
           return value;
           }
     void setValue(int value)
           {
            this.value = value;
           }
     }

class YourClass extends ABaseClass implements Serializable
     {
     //..
     }
```

---

# Custom serialization: Example code

You may want to use custom serialization if the base class does not support serialization and for some reason you don't like the default customization. Define your own `readObject()` and `writeObject()` methods. The fields or attributes are no longer automatically serialized if you define these methods. You have to take care of saving and restoring each of your attributes.

To store away the non-transient and non-static attributes for each object, call `defaultWriteObject()`. The attributes can be loaded with `defaultReadObject()`. You can store and load additional values in your method, for instance, ones that you have declared to be transient. If the class you derived from is serializable, then your data for the super class is already taken care of. All you have to do then is add your own custom code.

The example code below shows how `readObject()` and `writeObject()` are coded. Note that `ABaseClass` does not implement `Serializable`, but it does have `getValue()` and `setValue()`.

Here is the example code:

```
class ABaseClass
     {
     private int value;
     int getValue()
           {
           return value;
           }
     void setValue(int value)
           {
           this.value = value;
           }
     }

class YourClass extends ABaseClass implements Serializable
```

```
      {
      int another;
      private void readObject(ObjectInputStream ois)
            throws IOException, ClassNotFoundException
            {
             ois.defaultReadObject();
            setValue(ois.readInt());
            }

      private void writeObject(ObjectOutputStream oos)
            throws IOException
            {
            oos.defaultWriteObject();
            oos.writeInt(getValue());
            }
        }
```

---

# Externalizable: Example code

As you become more experienced with Java programming, you may decide you would like to customize the way things are done. You want to implement your own complete way of doing data persistence.

If you want to add your own custom code for storing away base class or superclass data, you must implement `Externalizable`. `Externalizable` has two methods: `readExternal()` and `writeExternal()`. In this case, you are fully responsible for everything.

Here is the example code:

```
 private void readExternal
      (ObjectInputStream ois)
      throws IOException, ClassNotFoundException
 private void writeExternal(
      ObjectOutputStream os)
      throws IOException
```

# Section 12. Tokenizing

## Introduction

This last stop on our journey introduces string tokenizing and covers the following topics:

* StringTokenizer
* StreamTokenizer

---

**StringTokenizer APIs:**

```
StringTokenizer(String string_to_tokenize,
  String delimiters)


boolean hasMoreTokens()
String nextToken()

String nextToken(String new_delimiters)

int countTokens()
```

## Using StringTokenizer

Remember earlier when you wrote lines that contain data items separated by delimiters? The delimiters, which separate the data items (or tokens, as they are called), might be commas or semicolons or tab characters.

When you read each line back in, you need to separate the tokens. Each line might represent a name and address, such as in a mail-merge file. We want to break that line up into its individual parts, such as name, street, city, state, and zip.

The way to do that is to use the `StringTokenizer` class. Although this is not part of the I/O hierarchy, we'll cover it here because it is useful in reading delimited streams. Here's how that class works.

---

## Constructing a StringTokenizer object

You construct an object of `StringTokenizer`, giving it the string you want to break up and what the delimiters are. That is, you tell it what characters are going to be used to break up the tokens in the string. After we have constructed an object of that type, we have a couple of things we can ask it:

* How many tokens are in this string?
* Have I used up all my tokens in this string?
* Give me the next token in the string.

`StringTokenizer` will break up the string and return to you as a `String` the characters up to the next delimiter. Its methods will throw a `NoSuchElementException` if there are no more tokens. Because this exception is derived from `RuntimeException` so that it is not declared with a throws clause, you do not need to catch it.

One more thing: you can switch delimiters for each token. In other words, even though you have created a `StringTokenizer` object that is looking for particular delimiters, you can say, "For this next token, change the delimiter."

The example on the next panel illustrates this process.

---

# StringTokenizer: Example code

Look at the code below. Notice our string has a vertical bar and a question mark. Those are our delimiters. We are going to create a new `StringTokenizer`, and we are going to pass it the string `abc (bar) def (question mark) ghi`. We are going to use as our delimiters a bar and a question mark.

The `StringTokenizer` `hasMoreTokens()` returns `true` if there are still more tokens. If it is `true`, we are going to call `nextToken()`. This returns a `String`, and `s` will have the value `abc` in it the first time around the `while` loop.

We come around the loop a second time and `hasMoreTokens()` is still true. Now when we get the next token, `def` is the value of the string that is returned.

Go around the loop a third time and `hasMoreTokens()` is still `true`; `ghi` is returned.

Now go around the loop one more time and `hasMoreTokens()` is `false`. At this point, we drop out of the loop. So we have broken up the string without too much effort.

Here is the example code:

```
import java.io.*;
import java.util.*;
public class StringTokenizerExample
   {
   public static void main(String args[])
      {
      String line = \"abc|def?ghi\";
      StringTokenizer st = new StringTokenizer(line, \"|?\");
      while (st.hasMoreTokens())
         {
         String s = st.nextToken();
         System.out.println(\"Token is \" + s);
         }
      }
   }
```

---

# StreamTokenizer

You can parse an entire input stream. Unlike the `StringTokenizer`, which parses a String, the `StreamTokenizer` reads from a `Reader` and breaks the stream into tokens. The parsing process is controlled by syntax tables and flags.

Each token that is parsed is placed in a category. The categories include identifiers, numbers, quoted strings, and various comment styles. The parsing that is performed is suitable for breaking a Java, C, or C++ source file into its tokens. `StreamTokenizer` is not in the I/O hierarchy, but because it is used with streams, we cover it here.

The `StreamTokenizer` recognizes characters in the range from u0000 through u00FF. Each character value can have up to five possible attributes. The attributes are white space, alphabetic, numeric, string quote, and comment character.

* A character that is *white space* is used to separate tokens.
* An *alphabetic* character is part of an identifier.
* A *numeric* character can be part of an identifier or a number.
* A *string quote* character surrounds a quoted string.
* A *comment* character precedes or surrounds a comment.
* A character that does not have any of these attributes is an ordinary character. When an ordinary character is encountered, it is treated as a single character token.

Flags can be set to alter the parsing. Line terminators can either be tokens or white space separating tokens. C-style and C++-style comments can be recognized and skipped. Identifiers are converted to lower case or left as is.

---

# Use of StreamTokenizer

To use a `StreamTokenizer`, you construct it with the underlying `Reader.` Then you set up the syntax tables and flags. Next, you loop on the tokens, calling `nextToken()` until it returns TT_EOF.

The `nextToken()` method parses the next token.The type is both returned by the method and also placed in the type field. The value of the token is placed in the `sval` field (`String` value) if the token is a word or in the `nval` field (`numeric` value) if the token is a number.

The token type can be either a character or a value, which represents the type of the token. If a token consists of a single character, the value of the type is the character value. If the token is a quoted string, the value is the value of the quote character. Otherwise it is one of the following:

* TT_EOF means the end of the stream has been read.
* TT_EOL means the end of the line has been read (if end of line characters are treated as tokens).
* TT_NUMBER means that a number token has been read.
* TT_WORD means that a word token has been read.

---

# Methods of StreamTokenizer

There are many methods in `StreamTokenizer` to set up the syntax tables. We'll only

mention two here. The first is `resetSyntax()`, which sets all characters to ordinary. The second is `wordChars()`, which gives a set of characters the alphabetic attribute.

---

# StreamTokenizer: Example code

This example is a simple one that shows the use of the `StreamTokenizer`. It breaks a file into words consisting of lower- or upper-case letters. When a word is found, `nextToken()` returns TT_EOF.

If an ordinary character is found (in this case, anything not set as alphabetic), `nextToken()` returns the value of that character.

```
import java.io.*;
import java.util.*;
public class StreamTokenizerExample
   {
   public static void main(String args[])
      {
      try
         {
         FileReader fr = new FileReader(t.txt);
         BufferedReader br = new BufferedReader(fr);
         StreamTokenizer st = new StreamTokenizer(br);
         st.resetSyntax();
         st.wordChars(\'A\', \'Z\');
         st.wordChars(\'a\', \'z\');
         int type;
         while ((type = st.nextToken()) != StreamTokenizer.TT_EOF)
            {
            if (type == StreamTokenizer.TT_WORD)
               System.out.println(st.sval);
            }
         }
      catch (IOException e)
         {
         System.out.println(e);
         }
      }
   }
```

# Section 13. Lab

## Setup

The purpose of this lab is to give you a chance to use the JDK on some real code.

If you worked through the topics, you're familiar with the code samples. The code was developed in Java. You can download the code samples and the JDK, then make your own modifications to the code. Then you'll recompile and run it. Let's begin the setup now. Make sure you are online so you can download the Java Development Kit. You may select any JDK, but we suggest selecting Java Development Kit 1.1 Platform (JDK 1.1). JDK 1.1 runs more programs than JDK 1.02 but is much smaller than JDK 1.2. Follow the instructions in the package to install the JDK.

In the gray scrolling text box, scroll down until you see the hypertext link to download the JDK. When you've finished downloading the JDK, come back to this part of the course to get the sample code and a database associated with the code.

*Download the JDK* .

If you already have the JDK, click Next to continue.

---

## Download Samples

Now let's *download the samples* . Move the archive file to your computer hard drive into a directory such as Java IO Lab. From your computer operating system, unpack the zip file into the Java IO Lab directory. In the directory you will see code sample files.

There are no special instructions with this lab. You may change the code any way you wish, then compile and run it.

# Section 14. Wrapup

## In summary

This rounds out our tour of the `java.io` package, a core API in the Java platform. We've covered a lot of ground in this tutorial. You've learned that input and ouput in the Java language is organized around the concept of streams, and that all input is done through subclasses of `InputStream` and that all output is done through subclasses of `OutputStream`. (Well, that's not quite true. You also learned about the one exception, class `RandomAccessFile`, which handles files that allow random access and perhaps intermixed reading and writing of the file.)

We've covered all the basics from buffering and filtering to checksumming, data compression, and serializing objects. We encourage you to reinforce this knowledge by working with the sample code until you feel comfortable.

---

## Resources

Learn more about the Java programming language and Java I/O from these resources:

* *The Java Tutorial: Essential Java Classes* has an excellent lesson on I/O.
* " *Java I/O* " by Elliotte Rusty Harrold (O'Reilly, 1999) is the quintessential guide to streaming in the Java language.
* Popular Java technology writer Todd Sundsted wrote an interesting article on `Readers` and `Writers`: *Use the two "R"s of Java 1.1 -- Readers and Writers* .
* Review the *java.io 1.1 package specification* .
* You may also want to review the *I/O specification for the Java 2 platform* .
* If you are relatively new to Java programming, the *Java language essentials* tutorial (developerWorks, November 2000) is an excellent resource.

---

## Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

---

### Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.