Input and Output

What Kind of Data can we Read or Write?

1. Text format (human readable)

```
# Today's Menu
pizza  320.00
coffee  20.00
```

2. Binary data

```
2A40FFFF89BB0101E9B70000
```

3. Special formats


TIFF PNG JPG

WAV AU

```
<?xml><title>In &amp; Out</title>
```

Read from **What** and **Where**?

1. Read from ***console*** or write to *console*:

```
InputStream in = System.in;
int b = in.read( );     // read one byte
System.out.print( b ); // write one byte
```

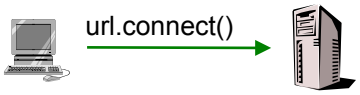
2. Read from a **file**:

```
InputStream in = new FileInputStream( "myfile.txt" );
int b = in.read( );
```

3. Read from another **process** or **thread** running on this computer.

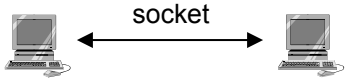
4. Read from a **known service** on this or remote machine.

```
URL url = new URL("http://www.ku.ac.th/index.html");
InputStream in = url.openStream( );
// read it
```



5. Read from the network using a **socket** (a network connection):

```
String host = "se.cpe.ku.ac.th";
int port = 17;
Socket socket = new Socket( host, port );
socket.connect( );
InputStream in = socket.getInputStream( );
```



6. Read from a **hardware device** such as USB, Gamepad, RFID reader

   Use Java Native Interface (JNI) or JInput to connect

   Special case: AudioDevice supported by Java Audio System

How To Read and Write?

Java has 3 levels of Input classes:

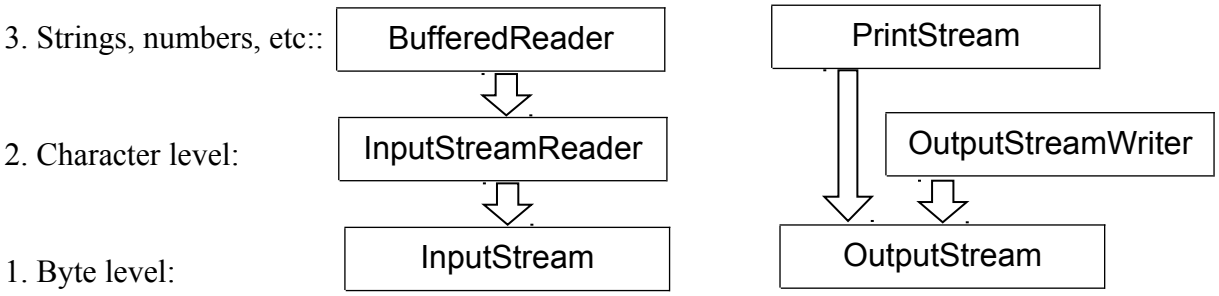
InputStream	read bytes
Reader	read characters, uses an InputStream to read bytes
BufferedReader	decorator for Reader. Reads lines as Strings.

And a matching hierarchy of Output classes:

OutputStream	write bytes
Writer	write characters, uses in OutputStream

Buffered writers writes Strings and other data structures

To avoid duplication, each class uses the services of a lower level class:



Examples

(1) Read a file as bytes

```
InputStream in = new FileInputStream( "somefile.txt" );
int b = in.read();
while( b >= 0 ) {    // read() returns -1 at end of file
   System.out.write( (byte)b );
   b = in.read( );
}
```

(2) Read a file as characters

```
InputStream in = new FileInputStream( "somefile.txt" );
Reader reader = new InputStreamReader( in );
int c = 0;
// reads Unicode character, returns -1 if no more input
while( (c = reader.read()) >= 0 ) {
   System.out.print( (char)c );
}
```

(3) Read a file line-by-line as strings

```
InputStream in = new FileInputStream( "somefile.txt" );
Reader reader = new InputStreamReader( in );
BufferedReader br = new BufferedReader( reader );
String line = null;
// readLine() is null when end of input is reached
while( (line = br.readLine()) != null ) {
   System.out.println( line );
}
```

InputStream.read( ) and InputStreamReader.read() return -1 when the end of input is reached. This is why they return int instead of byte or char.

Parsing Text Input

If the input contains character data (human readable text), then how can we read numbers, Strings, and other data types? Java provides a *scanner* to parse input values. java.util.Scanner reads an InputStream, parses it, and produces int, double, String, etc:

```
Scanner scanner = new Scanner( System.in ); // any input
stream
String s = scanner.next( ); // one word
if (scanner.hasNextInt() ) n = scanner.nextInt();
if (scanner.hasNextDouble() ) x = scanner.nextDouble( );
```

```
String line = scanner.nextLine( );
```

Scanner *parses* input at whitespace (space, tab, or newline) but you can change this. To parse the input at comma use `scanner.useDelimiter(",")`. useDelimiter accepts a *regular expression*. For example, to parse at comma with optional whitespace (`\s`) before and after the commas use: `scanner.useDelimiter("\\s*,\\s*")`.

Scanner is rather **fat** and **slow**. Applications that need faster parsing can use a BufferedReader to read lines and then split the line into strings. Before Java 5 this was the usual way of parsing input.

```
BufferedReader br = new BufferedReader(
                      new InputStreamReader(System.in) );
String line = br.readLine( );
String [] words = line.split("\\s+");
int n = Integer.parseInt( words[0] ); // first arg is int
```

StringTokenizer

StringTokenizer is a simple String parser that is faster than Scanner. It splits (tokenizes) a String at a given set of delimiters. If the token delimiter is complex, String.split( ) is more suitable since it uses regular expressions.
To split a String at comma characters using StringTokenizer:

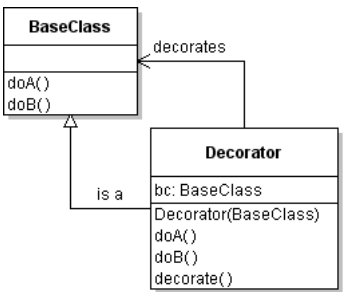
```
String s = "apple,banana,orange,grape";
StringTokenizer st = new StringTokenizer(s,",");
int count = st.countTokens();
while( st.hasMoreTokens() ) {
   System.out.println( st.nextToken() );
}
```

Formatting Text Output

To output numbers and other data types in character form (a human readable file), the values need to be converted to characters and formatted. PrintStream and PrintWriter provide this capability via methods like printf( ), println( ), and format( ). System.out is a PrintStream object.

```
PrintWriter writer = new PrintWriter( "myoutput.txt" );
writer.print( 2.5 ); // print method for primitive types
writer.println( "string with automatic end-line char" );
writer.printf( "%s %tT\n", "The time is now ",
                        System.currentTimeMillis());
```

Decorators, Filters, Adapters, and Wrappers



A Decorator is a class that adds extra functionality to another class without changing its interface. Decorators are used to add useful behavior to a class without making the underlying class more complex. The Decorator *extends* the base class (or *implements* a common interface) and *has an* (encapsulates) instance of the base class which it decorates. Decorators are also called *Filters* or *Wrappers*.

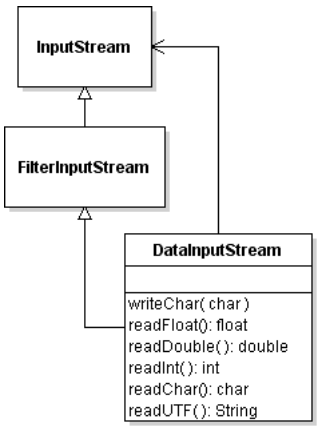
The Decorator passes most methods to the BaseClass object that it encapsulates (it would invoke bc.doA() .not super.doA()), or it may "enhance" those methods somehow.

Because the Decorator has the same interface as the BaseClass, we can substitute the Decorator in any application that is expecting an object of type BaseClass. Think of a decorator as *wrapping* a base object.

The Java I/O hierarchy has many decorators. For example, a PushbackInputStream decorates an InputStream by adding the ability to "unread" or "put back" bytes.

```
InputStream fin = new FileInputStream( "myfile.txt" );
PushbackInputStream pin = new PushbackInputStream( fin );
// you can use pin just as substitute for fin
InputStreamReader reader = new InputStreamReader( pin );
int b = reader.read( );
// read until we see a non-ASCII character
do {
   if ( b > 127)  { pin.unread(b); break; }
} while ((b = reader.read()) > 0);
```

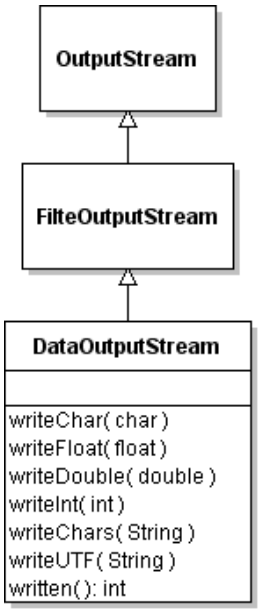
Reading Binary Data



Binary format is more compact than text format and I/O is usually faster. In binary format, an int is read as 4-bytes, long as 8-bytes, etc. Characters and Strings use modified UTF-8 format described in the *DataInput* interface. To read binary data, use a DataInputStream as a wrapper for an InputStream:

```
InputStream in = new
FileInputStream("song.wav");
DataInputStream din = new
DataInputStream( in );
byte flags = din.readByte( );
int size =  din.readInt( );
String title = din.readUTF( );
float f = din.readFloat( );
// skip some data
din.skipBytes( n );
// close the input when done
din.close();
```

Writing Binary Data



A DataOuputStream wraps an OutputStream and provides methods for writing data in binary format. Integers are written as 4 bytes, double at 8 bytes, etc. Characters are written as 2-byte UTF and Strings are written 2-bytes per character (writeChars) or using UTF format (writeUTF).

```
OutputStream out =
       new FileOutputStream("mydata.dat");
DataOutputStream dout = new DataOutputStream(out);
dout.writeInt( 10 );
dout.writeDouble( Math.PI );
String s = "hello";
dout.writeInt( s.size() );
dout.writeChars( s );
// how many bytes written?
count = dout.written( );
dout.close( );
```

Common I/O Operations

In addition to reading and writing data, I/O classes need to provide some basic functionality:

close( )

Close the input or output resource. Closing input/outstreams frees system resources. For output streams, it ensures that all the data is written before the application terminates. If the reference is a wrapper, then calling close should close the underlying stream. Most I/O classes have a close method.

End of Input?

Input classes should provide a way for the user to determine *when* the end of input has been reached. InputStream and InputStreamReader return -1 if read( ) encounters the end of input. BufferedReader.readLine returns a null String at the end of input. A DataInputStream throws an EOFException when the end of input is encountered.

Append or Overwrite?

If you open a file for *output* (writing) and the file already exists, the default is to *replace* what was in the file -- destroying the old file contents. The I/O framework must give the programmer a way to *append* to an existing file. FileOutputStream and FileWriter have constructors with a second parameter, which is a flag for append:

FileOutputStream(String filename, boolean append).

For example, if your application creates a *log file* you probably want to *append* new information to the log file and not delete old information.

```
String filename = "myapp.log";
boolean append  = true;
OutputStream log = new FileOutputStream(filename, append);
```

Buffering and flush()

Reading and writing data one byte at a time is inefficient. Files on the file system cannot be written one byte at a time, since they are not byte-addressable. Most I/O classes, therefore, store some data in memory called a *buffer*. Even classes without "buffer" in their name may buffer data.

A programmer needs to understand buffering because it affects the way some methods behave. Here are two examples:

1. Reading an array does not read fully. InputStream can read an array of bytes. It returns the number of bytes read. For example:

```
byte [] data = new byte[80];
// this methods reads bytes into an array.
int count = inputStream.read( data );  // may not be 80
```

count is the number of bytes *actually read*. It is -1 if no more data. Many programmers assume that if there is lots of data then inputStrream.read() will completely fill the array. In this example they would assume 80 bytes were read. But this is not correct! If inputStream has some data in its own *buffer*, it may return only that data and not read anything the underlying source until the buffer is empty. You should check the return value (count) for amount of data actually read.

2. Buffered Output. An OutputStream may not *immediately* write output to the destination (such as a file or network). It may *buffer* data temporarily. PrintStream and other classes *buffer* output for efficiency. Here is an example: after writing to a file, the data does not appear in the file.

```
OutputStream out = new FileOutputStream( "myfile.txt" );
PrintStream writer = new PrintStream( out );
```

```
writer.print("Hello world");
   // nothing is written in the file yet!
```

To force the output to be written *immediately* to the file, invoke flush( ).

```
writer.print("Hello world");
   // nothing is written in the file yet!
writer.flush( );
   // now the output is written to file
```

Reading and Writing Objects

You can save (write) and recreate (read) objects using an ObjectOutputStream and ObjectInputStream. These classes *serialize* an object's data (but not its methods). A class must declare that it implements *Serializable* for its objects to be serialized. Serializable is a "marker" interface -- you don't have to implement any methods, but you *should* declare a static final serialVersionUID, and increment this number each time you change the fields in the class:

```
public class Person implements java.io.Serializable {:
   public static final long serialVersionUID = 11;
   private String name;
   private Date birthday;
   ...
```

To save a Person, including all the objects it references (name and birthday) to a file use:

```
Date birthday  = new Date(95,2,15);
Person person = new Person( "Mr. Java", birthday);
OutputStream out = new FileOutputStream("person.data");
                          // any filename is ok
ObjectOutputStream ostream = new ObjectOutputStream( out );
ostream.writeObject( person );
ostream.flush( ); // force object to be written
```

ObjectOutputStream also has methods like writeInt( ) and writeChar( ) for writing primitives, characters, and Strings; these methods are the same as DataOutputStream.

To recreate an Object from serialized data, use an ObjectInputStream. To read the Person data saved in the above example use:

```
InputStream in = new FileInputStream("person.data");
ObjectInputStream istream = new ObjectInputStream( in );
Person x = (Person) istream.readObject( );
```

It is the programmer's responsibility to ensure that the file contains object data for the desired type of object, and that the class has not changed since the object was saved (same serialVersionUID). Otherwise, it will throw an exception.

Handling Exceptions

The input and output methods throw lots of exceptions. The most common ones are:

IOException - an error occurred while reading or writing

FileNotFoundException - the requested file was not found *or* the process does not have permission to access the file (read or write permission). This is a *subclass* of IOException, so a catch block for IOException will also catch FileNotFoundException.

EOFException - Thrown by *some* input classes when end of input is encountered while reading.

To read a file and then close it after reading or after an exception occurs, this code is typical:

```
FileInputStrream in = null;
try {
   in = new FileInputStream( filename );
   int b;
   while ((b = in.read()) >= 0) /* process the input */;
} catch (FileNotFoundException fne) {
   System.err.println("Could not access file "+filename );
} catch (IOException ioe) {
   System.err.println( ioe.getMessage() );
} finally {
   // always close the file
   if (in != null) try {
      in.close( );
   } catch(IOException ex) { /* ignore it */ }
}
```

What is the File class? Why use it?

The java.io.File class is used to represent files or directories. File objects can be used to get informatiion about a file, to find all files in a directory, and as argument to other I/O classes that read/write files.

```
File file = new File("mydata.txt");
if ( file.exists() ) System.out.println("File exists.");
if ( file.canRead() ) System.out.println( "You can read
it.");
if ( file.isFile() ) System.out.println( "Its an ordinary
file" );
if ( file.isDirectory() ) System.out.println("Its a
directory");
String path = file.getCanonicalPath( );
System.out.println("Fully qualified filename is " +path);
```

The File class can also be used to create a *temporary file* in whatever directory the current system uses for temporary files.

```
File tempfile = File.createTempFile("autosave", "tmp");
```

Most classes that create input/output streams for files have 2 constructors: one constructor accepts a String filename and another accepts a File object. You can use a File object to first test if I/O will succeed, then use the File object to create a FileInputStream or FileOutputStream.

One example use is to avoid overwriting an existing file. Suppose the user inputs a (String) filename for an output file. Your application can check if the file already exists before opening it for writing:

```
File file = new File( filename );
if ( file.exists() ) return false; // don't overwrite
FileOutputStream out = new FileOutputStream( file );
```

Using Network Resources

You can read or write to resources on the network using the same kinds of InputStream and OutputStream you use to write to a local file or console. We will look at these cases:

1. I/O using a URL for a known protocol

2. I/O using a socket for raw data or "do it yourself" service.

Uniform Resource Locator (URL)

The above describes how to read/write using resources on the local computer. You can also read and write to resources on the network -- using the same InputStream, OutputStream, and wrappers (decorators) as input/output to a local file.

A standard syntax for identifying network resources is the Uniform Resource Location (URL). Some examples of URL are:

- a file on a web server or web service: `http://www.ku.ac.th/index.html`
- a file on an FTP server: `ftp://ftp.somehost.com/pub/file.txt`
- web server or web service on port 8080:
  `http://se.cpe.ku.ac.th:8080/someapp/file.txt`

The general format for a URL is:

`protocol://`*`hostname`*`[:`*`port`*`]/path/resource`

You can create an InputStream and read data from a URL just like any other InputStream:

```
import java.net.URL;

String urlstring = "http://se.cpe.ku.ac.th/alice.txt";
try {
   URL url = new URL( urlstring );
   InputStream instream = url.openStream( );
//TODO: read from instream

   instream.close();
}
catch (MalformedURLException mfe) { /* handle it */ }
catch (IOException ioe) { /* handle it */ }
```

You should close a URL when you are done with it. This is a good place to use a "finally" block.

Sockets

You can read *raw* data from a local or remote process using a Socket. Sockets are a standard interface provided by many programming languages. Java provides a general Socket and a ServerSocket that makes it easy to write a server-side Socket application.

To use a client-side socket, you need a *hostname* or *IP address* of a server, and a *port number*.

For example:

```
   String host = "se.cpe.ku.ac.th";
   int    port = 17;  // quote of the day service
   Socket socket = null;
   try {
      socket = new Socket( host, port );
      InputStream in = socket.getInputStream();
//TODO read data from the socket using InputStream in

   }
   catch( UnknownHostException uhe ) { /* print it */ }
   catch( IOException ioe ) { /* print it */ }
   finally {
      // close the socket
      if (socket != null && ! socket.closed() ) {
```

```
            try { socket.close(); }
            catch(IOException ex) { /* log it */ }
        }
    }
```