Streams

- Java provides many stream classes that let you work with data in the forms that you usually use rather than at the low, byte level.
- These are implemented in the abstract classes `InputStream` and `Outputstream`.
- The methods in these classes provide the ability to do simple, byte oriented operations.

Streams

- To receive information, a program opens a stream and reads the information:

- A program can send information by opening a stream to a destination and writes the informat

Using Streams

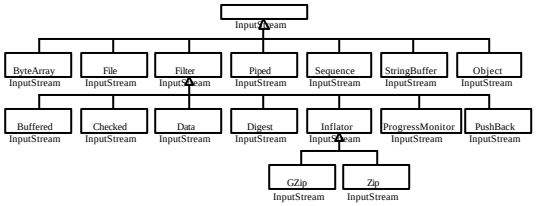
- No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data is pretty much always the same

```
Reading                        Writing
   open a stream                  open a stream
   while more information         while more information
       read information               write information
   close the stream               close the stream
```

java.io.*

- The java.io package contains a collection of stream classes that support reading/writing from/to streams
- Streams are divided into two class hierarchies based on the type of data on which they operate.

InputStream Classes



Data Sink Streams

Sink Type	Character Streams	Byte Streams
Memory	CharArrayReader CharArrayWriter StringReader StringWriter	ByteArrayInputStream ByteArrayOutputStream
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream
File	FileReader FileWriter	FileInputStream FileOutputStream

Data Processing Streams

Process	Character Streams	Byte Streams
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	FilterReader, FilterWriter	FilterInputStream, FilterOutputStream
Converting between bytes and characters	InputStreamReader OutputStreamReader	
Concatenation		SequenceInputStream
Object Serialization		ObjectInputStream, ObjectOutputStream
Data Conversion		DataInputStream, DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking Ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

Streams and Files

- Data files are handled using two abstractions:
  - the basic file abstraction is provided by the library class `File`. This encapsulates all the details of what a file is and how it is named.
  - The stream abstraction provides a way of reading and writing data to and from a file.
- Reader streams deal with text input and are subclasses of `Reader`, writer streams perform text output and are subclasses of `Writer`.

BufferedReader

- A `BufferedReader` reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- The buffer size may be specified, or the default size may be used.
- A `BufferedReader` is usually wrapped around any `Reader` whose `read()` operations may be costly.

FileEcho

```
import java.io.*;

public class FileEcho {
  public static void main( String args[] ) {
    int ch;

    if ( args.length>0 ) {
      BufferedReader  in = null;

      try {
        in = new BufferedReader ( new FileReader( args[0] ) );

        while ( ( ch = in.read() ) != -1 ) {
          System.out.print( (char)ch );
        }
      }
      catch ( FileNotFoundException e ) {
        System.out.println( "File not found" );

      }
      catch( IOException e ) {
        System.out.println( "Read error" );
        System.exit(1);
}}}}
```


InputEcho

```
import java.io.*;

public class InputEcho {
  public static void main( String args[] ) {
    // Set things up to read from the keyboard

    BufferedReader
      keyboard = new BufferedReader( new InputStreamReader( System.in ) );

    // Read stuff form input and dump to output

    try {
      String inString;

      while ( ( inString = keyboard.readLine() ) != null ) {
        System.out.println( inString );
      }
    } catch ( IOException e ) {
        System.err.println( "InputEcho:  I/O error" );
        System.exit( 1 );
    }
  }
}
```


ReadNums

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

public class ReadNums {
  public static void main( String args[] ) {
    // Make sure the number of arguments is correct

    if ( args.length != 1 ) {
      System.err.println( "Usage:  ReadNums sourceFile" );
      System.exit(1); }

    // Initialize src since the assignment is done inside a try block

    BufferedReader src = null;

    // Attempt to open the file for reading

    try {
      src = new BufferedReader( new FileReader( args[0] ) );
    }
    catch ( FileNotFoundException e ) {
      System.err.println( "ReadNums:  Unable to open source file" );
      System.exit(1); }
```

ReadNums (*continued*)

```
// Read the numbers a line at a time from the source file

Vector data = new Vector();

try {
  String line;

  while ( ( line = src.readLine() ) != null ) {
    try {
      int num = Integer.parseInt( line );
      data.addElement( new Integer( num ) );
    }
    catch ( NumberFormatException e ) {} }

  src.close();
}
catch ( IOException e ) {
  System.err.println( "ReadNums:  " + e.getMessage() );
  System.exit(1); }

// Print out the results
for ( int i=0; i<data.size(); i++ )
  System.out.println( data.elementAt( i ) ); }}
```


InputStreamReader

- An `InputStreamReader` is a bridge from byte streams to character streams: it reads bytes and translates them into characters according to a specified character encoding.
- Each invocation of one of an `InputStreamReader`'s `read()` methods may cause one or more bytes to be read from the underlying byte-input stream.

PrintWriter

- A `PrintWriter` prints formatted representations of objects to a text-output stream.
- Flushing does not occur until the `flush()` method is invoked. It is possible to enable automatic flushing, which causes a flush to take place after any `println()` method is invoked. The output of a newline character does not cause a flush.
- Methods in this class never throw I/O exceptions.

Building a Simple Expression Evaluator

- Consider building an expression evaluator for the following:
  - *identifier number* **;**
  - *identifier* **;**
- The code to do this follows. It uses the following classes:
  - `InputStreamReader`
  - `StreamTokenizer`
  - `HashTable`

ExprEval

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

public class ExprEval {
  public static void main( String args[] ) {
    StreamTokenizer lex =
      new StreamTokenizer( new InputStreamReader( System.in ) );

    final int WANT_WORD = 0;
    final int WANT_NUM = 1;
    final int WANT_SEMI = 2;

    int curState = WANT_WORD;
    String curId = null;

    Hashtable symTbl = new Hashtable();

    lex.ordinaryChar( '-' );    // '-' is treated normally
    lex.lowerCaseMode( true );  // make case insensitive
```


ExprEval

```
    try {
      while ( lex.nextToken() != StreamTokenizer.TT_EOF ) {
        switch ( lex.ttype ) {
          case lex.TT_WORD:
            if ( curState == WANT_WORD ) {
              curId = lex.sval; curState = WANT_NUM;
            }
            else
              curState = WANT_SEMI;
            break;
          case lex.TT_NUMBER:
            if ( curState == WANT_NUM )
              symTbl.put( curId, new Integer( (int)lex.nval) );
            curState = WANT_SEMI; break;
          case ';':
            if ( curState == WANT_NUM )
              System.out.println( symTbl.get( curId ) );
            curState = WANT_WORD; break;
          default:
            curState = WANT_SEMI;
  }}} catch( Exception e ) { System.out.println( "I/O Error" ); }}}
```