The Java I/O System

The Java I/O System

- Introduction
- I/O streams
- Byte I/O streams
- Character I/O streams
- The Decorator pattern
- Buffered streams
- Useful I/O classes
- NIO

Introduction

- Bruce Eckel on Java I/O:
  - “I can't remember how to open files in Java. I've written chapters on it. I've done it a bunch of times, but it's too many steps. And when I actually analyze it, I realize these are just silly design decisions that they made. […] They should have had a convenience constructor for opening files simply. Because we open files all the time, but nobody can remember how. It is too much information to hold in your mind.”

“Python and the Programmer – A Conversation with Bruce Eckel, Part I”

http://www.artima.com/intv/aboutme2.html

Why is Java I/O complicated?

- Creating a good I/O system is one of the more difficult tasks for the language designer:
  - There are different sources and sinks of I/O that you want to communicate with – files, the console, pipes, network connections, etc.
  - You need to talk to them in a wide variety of ways: sequential, random-access, buffered, binary, character, by lines, by words, etc.
- The Java library designers attacked this problem by creating lots of classes
- Any given kind of I/O is not particularly difficult: the trick is to find your way through the maze of possibilities

The Java I/O System

- Introduction
- I/O streams
- Byte I/O streams
- Character I/O streams
- The Decorator pattern
- Buffered streams
- Useful I/O classes
- NIO

I/O Streams

- An *I/O stream* represents an input source or an output destination
  - Distinct (and older) from the Java 8 stream API
- A stream can represent many different kinds of sources and destinations – files, network connections, memory locations, etc.
- It also supports many different kinds of data – simple bytes, primitive data types, localized characters, or objects
- Some streams simply pass on data; others manipulate and transform the data in useful ways
- But, no matter how they work internally, all streams present the same simple model to programs that use them
  - To a program, a stream is always a sequence of data

Java I/O streams

- All Java I/O streams, as well as supporting classes, are located under the *java.io* package
- An I/O stream is unidirectional: it can be created for either reading or writing the data, but not both
- Java I/O streams can be grouped in several ways:
  - Byte and character streams
  - Input and output streams
  - Node and filter streams

Byte and character streams

- Byte streams operate on raw, 8-bit bytes; they offer methods for reading or writing a single byte, or an array of bytes
  - Root classes for byte streams are *InputStream* and *OutputStream*
- Character streams, on the other hand, provide Unicode-compliant, character-based I/O
  - Root classes for character streams are *Reader* and *Writer*
- *All 4 classes are abstract*

Input and output streams

- Input or source streams are used to read data
  - Root classes for input streams are *InputStream* and *Reader*
- Output or sink (or destination) streams are used to write data
  - Root classes for output streams are *OutputStream* and *Writer*

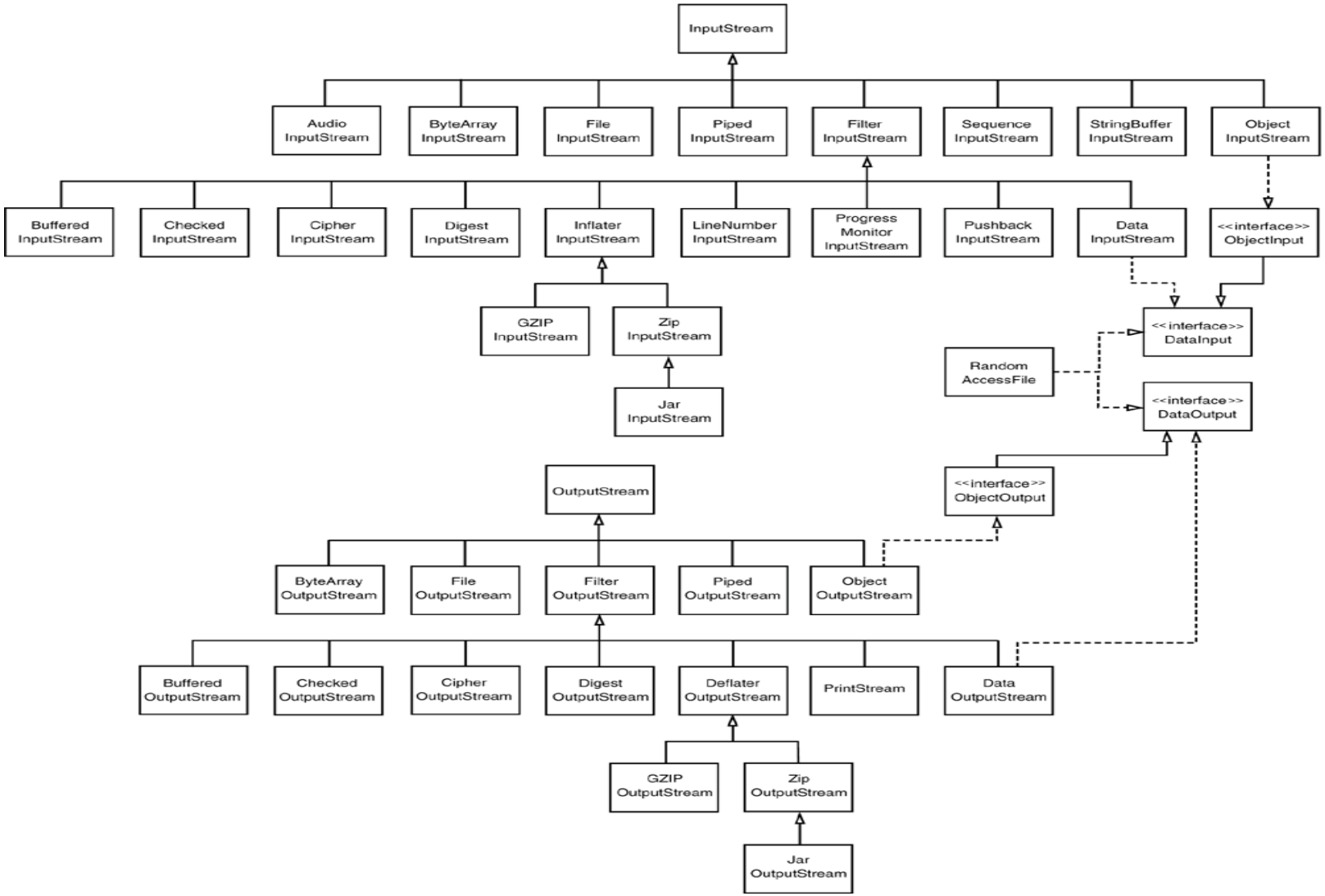
Node and filter streams

- Node streams are low-level streams, containing the basic functionality of reading from or writing to memory locations, files, etc.
- Filter or processing streams and layered onto node streams for additional functionality – altering or managing data in the stream
  - Example: *ZipInputStream* for reading files in the *ZIP* file format

The Java I/O System

- Introduction
- I/O streams
- Byte I/O streams
- Character I/O streams
- The Decorator pattern
- Buffered streams
- Useful I/O classes
- NIO

Byte I/O streams



The *InputStream* abstract class

- *InputStream* is the superclass of all classes representing an input stream of bytes
- Its most important methods include the following:
  - **public abstract int read()**: reads the next byte of data (a value in the range 0 to 255) from the input stream; if no byte is available because the end of the stream has been reached, the value -1 is returned
  - **public int read(byte[] b)**: reads some number of bytes from the input stream into *b*; the number of bytes actually read is returned as an integer
  - **public int read(byte[] b, int off, int len)**: reads up to *len* bytes of data from the input stream into *b* starting at position *off*
- All methods throw *IOException* if some I/O error occurs
- In addition, all methods block until input data is available, end of file is detected, or an exception is thrown

The *InputStream* abstract class (cont')

- To avoid blocking, use the **public int available()** method
  - The method returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream
- The **public long skip(long n)** method can be used to skip over a number of bytes
  - The actual number of bytes skipped is returned – it may, for a variety of reasons, end up skipping over some smaller number of bytes, possibly 0
  - Note that a stream may not have the ability to seek, in which case the method throws an exception

Node *InputStream* classes

- *FileInputStream*: meant for reading streams of raw bytes, such as image data, from a file
- *ByteArrayInputStream*: wrapper around an internal buffer that contains bytes that may be read from the stream
  - The class is used when a stream-like behavior is required from an array of bytes
- *ObjectInputStream*: used to read objects from the stream, during the process of *deserialization*
- *AudioInputStream*: an input stream with a specified audio format and length; the length is expressed in *samples* not bytes
- *Note: the list presents only a subset of all node InputStream classes*

Node *InputStream* classes – example

- The following example shows how to process raw bytes from a file, one by one

```
public static void main(String[] args) {
  FileInputStream file = null;
  try {

    file = new FileInputStream("binaryfile.dat");
    int data;
    while ((data = file.read()) != -1)
      process(data);

  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    if (file != null)
      try {
          file.close();
      } catch (IOException e) { }
  }
}
```

Closing streams

- Every I/O stream exposes method *close()* for closing the stream
- Closing a stream when it's no longer needed is very important – so important that the program should use a *finally* block to guarantee that the stream will be closed even if an error occurs
  - This practice helps avoid resource leaks or data loss
- The trouble is that the *close()* method might generate *IOException*, e.g. if the program tries to close a file that has already been closed
  - This is why there needs to be a *try … catch* block inside the *finally* block
- An alternative is the *try with resources* statement (Java 7)

Filter *InputStream* classes

- All filter input streams extend the *FilterInputStream* class (a direct descendant of *InputStream*)
  - These streams transform the basic source data along the way
- *BufferedInputStream*: adds the ability to buffer the input
- *DataInputStream*: lets an application read primitive Java data types from an underlying input stream in a machine-independent way
  - It offers methods for reading *int, float, double*, etc. values
- *CheckedInputStream*: maintains a checksum of the data being read; the checksum can then be used to verify the integrity of the input data

The *OutputStream* abstract class

- *OutputStream* is the superclass of all classes representing an input stream of bytes
- Its most important methods include:
  - **public abstract void write(int b)**: writes the specified byte to this output stream; more precisely, it writes the eight low-order bits of *b*, while the 24 high-order bits are ignored
  - **public void write(byte[] b)**: writes *b.length* bytes from the specified byte array to this output stream
  - **public void write(byte[] b, int off, int len)**: writes *len* bytes from the specified byte array starting at offset *off* to this output stream
- All methods throw *IOException* if some I/O error occurs

OutputStream classes

- Node and filter *OutputStream* class descendants are counterparts of *InputStream*-based classes
- Example node *OutputStream* classes include:
  - *FileOutputStream*
  - *ByteArrayOutputStream*
  - *ObjectOutputStream* (used to write objects to the stream, during the process of *serialization*)
- Example filter *OutputStream* classes include:
  - *BufferedOutputStream*
  - *DataOutputStream*
  - *CheckedOutputStream*

OutputStream classes – example

- The following example shows how to transfer data from one file to another, up to *SIZE* bytes at a time

```
public static void main(String[] args) {
  FileInputStream input = null;
  FileOutputStream output = null;
  try {
    // open files
    input = new FileInputStream("input.dat");
    output = new FileOutputStream("output.dat");
    // maximum data size to be read
    final int SIZE = 1024;
    byte[] data = new byte[SIZE];
    int sizeRead; // how many was actually read
    while ((sizeRead = input.read(data)) != -1) {
      // process only the first sizeRead bytes
      process(data, sizeRead);
      // write only the first sizeRead bytes
      // invoking "output.write(data)" would write
      // the entire array
      output.write(data, 0, sizeRead);
    }
```

OutputStream classes – example (cont')

```
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      // close the input file (if it was open)
      if (input != null)
        try {
          input.close();
        } catch (IOException e)
        {
        }
      // close the output file (if it was open)
      if (output != null)
        try {
            output.close();
        } catch (IOException e)
        {
        }
    }
}
```

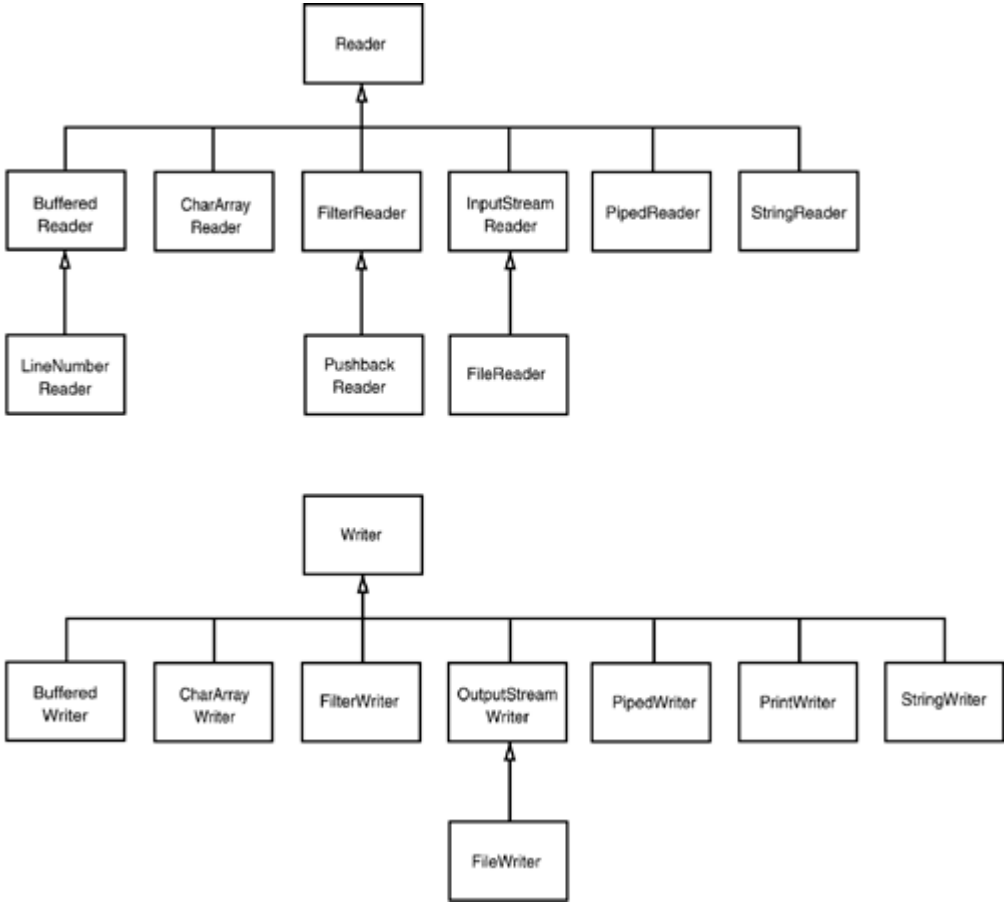
Note on byte streams

- Byte streams represent a kind of low-level I/O
- They are all used in much the same way – they differ mainly in the way they are created
- All other streams are based on byte streams
- Byte streams should only be used when operating on the raw binary data
  - That is, if the data contains characters, the best way is to use character streams

The Java I/O System

- Introduction
- I/O streams
- Byte I/O streams
- Character I/O streams
- The Decorator pattern
- Buffered streams
- Useful I/O classes
- NIO

Character I/O streams



Benefits of character streams

- Character I/O streams automatically translate internal data format to and from the local or specified character set
- All character stream classes are descended from *Reader* and *Writer*
- For most applications, I/O with character streams is no more complicated than I/O with byte streams
- In fact, each method of the *Reader* class has a corresponding method in *InputStream,* with one difference: methods of the *Reader* class use *char* instead of *byte* as the main type
  - The same is true for *Writer* and *OutputStream* classes

Using character streams – example

- The following example demonstrates how automatic conversion of raw bytes into local character set is performed
  - The *FileReader* class uses the system's default character encoding, which has been set to UTF-8

```
public static void main(String[] args) {
  Reader reader = null;
  try {

    reader = new FileReader("C:\\sample.txt");
    StringBuilder builder = new StringBuilder();

    int i;
    while ((i = reader.read()) != -1)
      builder.append((char)i);
```

Using character streams – example (cont')

```
    System.out.println(builder);

    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (reader != null)
        try {
          reader.close();
        } catch (IOException e) { }
    }
}
```

- Raw file content (UTF-8 encoded Cyrillic text):

```
ÐĄÐµÐ»Ð»ÐĬ ÐŠÐĬŃ€Ð»Ð´!
```

- Program output:

```
Хелло Њорлд!
```

Character streams as “wrappers”

- Character streams are often “wrappers” for byte streams
- The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes

Byte stream	*Character stream “wrapper”*
InputStream	Reader
OutputStream	Writer
FileInputStream	FileReader
FileOutputStream	FileWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter

- There are two general-purpose byte-to-character “bridge” streams: *InputStreamReader* and *OutputStreamWriter*
  - These are used to create new character streams, i.e. when there are no prepackaged character stream classes

InputStreamReader example 1

- The standard input stream, which typically corresponds to keyboard input, is represented by the *System.in* field of type *InputStream*
- To read characters from the standard input stream, however, one needs to create a character stream wrapper

```
public static void main(String[] args) throws IOException {
  Reader reader = new InputStreamReader(System.in);
  char[] text = new char[10];
  System.out.print("Input? ");
  reader.read(text);
  System.out.println("You've entered: " + String.valueOf(text));
}
```

- Console:

```
Input? Hello World!
You've entered: Hello Worl
```

InputStreamReader example 2

- As noted before, the *FileReader* class uses the system's default character encoding – whatever that might be
- And there is no way to change this behavior
  - If the file was encoded using UTF-16 character set, reading it with *FileReader* would produce "garbage"
- This is why *FileReader* should be used with care
- A more flexible way is to use *InputStreamReader* on top of *FileInputStream* and set a custom encoding scheme

InputStreamReader example 2 (cont')

```
public static void main(String[] args) throws IOException {
  Reader reader = null;
  try {

    // use the UTF-16 encoding
    reader = new InputStreamReader(
      new FileInputStream("C:\\sample.txt"), "UTF-16");
    StringBuilder builder = new StringBuilder();

    int i;
    while ((i = reader.read()) != -1)
      builder.append((char)i);
    System.out.println(builder);

  } finally {
    if (reader != null)
      reader.close();
  }
}
```

- Output: `Хелло Њорлд!`
- Output of the program using *FileReader*: `??%□ 5□ ;□ ;□ >□`

The Java I/O System

- Introduction
- I/O streams
- Byte I/O streams
- Character I/O streams
- The Decorator pattern
- Buffered streams
- Useful I/O classes
- NIO

The *Decorator* pattern

- In the last example, the statement:

```
reader = new InputStreamReader(
  new FileInputStream("C:\\sample.txt"), "UTF-8");
```

  shows how additional functionality can be added to an existing object, by stacking another layer onto it
  - *FileInputStream* returns raw bytes, and then *InputStreamReader* transforms them into characters

- In the world of OO design, the use of layered objects to dynamically and transparently add responsibilities to individual objects is referred to as the *Decorator pattern*

- Decorators are often used when simple subclassing results in a large number of classes in order to satisfy every possible combination that is needed
  - So many classes that it becomes impractical

The *Decorator* pattern drawback

- The *Decorator* pattern adds much more flexibility while writing a program, since attributes can easily be mixed and matched as needed
- However, there is a drawback to this approach: it adds complexity to the code
- The reason that the Java I/O library is awkward to use is that you must create many objects in order to get the single I/O object that you want

The Java I/O System

- Introduction
- I/O streams
- Byte I/O streams
- Character I/O streams
- The Decorator pattern
- Buffered streams
- Useful I/O classes
- NIO

Buffered streams

- An unbuffered I/O means that each read or write request is handled directly by the underlying OS
- This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive
- To reduce this kind of overhead, the Java platform implements buffered I/O streams
- Buffered input streams read data from a memory area known as a *buffer* – the native input API is called only when the buffer is empty
- Similarly, buffered output streams write data to a buffer and invoke native output API only when the buffer is full

Creating a buffered stream

- With the help of the *Decorator* pattern, developers can easily create buffered streams: by layering a buffered stream onto an unbuffered one
- For byte streams, the buffered classes to use are *BufferedInputStream* and *BufferedOutputStream*
- For character streams, the buffered classes to use are *BufferedReader* and *BufferedWriter*
  - Besides being more efficient than an un-buffered reader, the *BufferedReader* class adds a convenient *public String readLine()* method
- Each of these classes has a constructor that receives an object of the underlying unbuffered stream

Flushing a buffered output stream

- Note that when writing to a buffered output stream, the data is saved into the buffer, and actually written out only when the buffer is full
- It often makes sense to write out a buffer at critical points, without waiting for it to fill – this is known as *flushing* the stream
- To flush a stream manually, invoke its *flush()* method
  - Although the method is declared in *OutputStream* and *Writer* classes, it has no effect unless the stream is buffered
- Some streams support *auto-flushing* which can be activated by an optional constructor argument
  - When auto-flush is enabled, certain key events, such as the newline character, cause the buffer to be flushed

The Java I/O System

- Introduction
- I/O streams
- Byte I/O streams
- Character I/O streams
- The Decorator pattern
- Buffered streams
- Useful I/O classes
- NIO

Useful I/O classes

- The following section analyses some most commonly-used, powerful classes for:
  - Abstract, system-independent representation of files and directories
  - Writing character output to various sinks
  - Reading input from various sources

The *File* class

- The *File* class is an abstract, system-independent representation of hierarchical pathnames
  - It represents both files and directories (list of files)
- An abstract pathname has two components:
  - An optional system-dependent prefix string, such as a disk-drive specifier
  - A sequence of zero or more string names: each name except for the last denotes a directory, whereas the last name may denote either a directory or a file
- Each name is separated from the next by a single copy of the default separator character
  - The separator is made available in the public static fields *separator* and *separatorChar* of the class

The *PrintWriter* class

- The most convenient and efficient way for writing character output is to use the *PrintWriter* class
- This is because:
  - The class is buffered, and supports auto-flushing
  - It offers a number of methods for writing various data types, lines, formatted strings, etc.
  - It offers several convenience constructors that receive *OutputStream*, *Writer*, or a file
  - Its methods don't throw exceptions; instead they set an internal flag which can be queried through *checkError()*
- Note that *PrintWriter* performs auto-flushing only when one of the *println()*, *printf()*, or *format()* methods is invoked

PrintWriter – auto-flush example

```
public static void main(String[] args) {
  // true means turn on auto-flushing
  PrintWriter writer = new PrintWriter(System.out, true);

  writer.println("Line number 1."); // auto-flush
  System.out.println("Line number 2.");

  writer.print("Line number 3.");   // written to buffer
  System.out.println("Line number 4.");
  writer.flush();
}
```

- Output:

```
Line number 1.
Line number 2.
Line number 4.
Line number 3.
```

PrintWriter – writing to a file example

```
public static void main(String[] args) {
  PrintWriter writer = null;
  try {

    // the constructor might still throw an exception
    // e.g. if the file cannot be created
    writer = new PrintWriter("C:\\output.txt");
    ...
  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    if (writer != null)
      writer.close(); // doesn't throw an exception
  }
}
```

The *Scanner* class

- *Scanner* represents a simple text scanner which can parse primitive types and strings using regular expressions
- Similarly to the *split()* method of *String* class, a *Scanner* breaks its input into tokens using a delimiter pattern, which by default matches whitespace
- However, the *Scanner* class is more powerful as it:
  - Automatically performs data conversion, using the various *next* methods
  - It can read text from a *File*, *InputStream*, *String*, or *ReadableByteChannel* object

Methods of the *Scanner* class

- **boolean hasNext()**: *true* if this scanner has another token in its input
- **boolean hasNextX()**: *true* if the next token in this scanner's input can be interpreted as a value of type *X*, with the *X* being any primitive data type
- **String next()**: returns the next complete token from this scanner
- **X nextX()**: scans the next token of the input as an *X*, with the *X* being any primitive data type
- **String nextLine()**: advances this scanner past the current line and returns the input that was skipped
  - Note that all *next*()* methods throw an exception if no more tokens are available
- **Scanner useDelimiter(String pattern)**: sets this scanner's delimiting pattern to a pattern constructed from the specified string

The *Scanner* class – example

```
public static void main(String[] args) {
  // input file name
  Scanner in = new Scanner(System.in);
  File file = new File(in.nextLine());

  if (file.exists()) {
    Scanner fin = null;
    try {
      // open the file and exclude all non-digits
      fin = new Scanner(file);
      fin.useDelimiter("[^0-9]+");

      System.out.println("The numbers are:");
      while (fin.hasNextInt())
        System.out.println(fin.nextInt());

    } catch (IOException e) {
      e.printStackTrace();
    } finally { // need to close the file
      if (fin != null)
        fin.close();
    }
  }
}
```

The *Scanner* class – example (cont')

- File content:

```
The file contains numbers 12 and 46, and
also the number 13
```

- Console:

```
Z:\tmp\Readme.txt
The numbers are:
12
46
13
```

The Java I/O System

- Introduction
- I/O streams
- Byte I/O streams
- Character I/O streams
- The Decorator pattern
- Buffered streams
- Useful I/O classes
- NIO

NIO

- The traditional I/O abstractions of the Java platform have served well and are appropriate for a wide range of uses
- However, these classes have a couple of drawbacks:
  - They do not scale well when moving large amounts of data
  - They do not provide some common I/O functionality widely available on most operating systems today, such as file locking, non-blocking I/O, memory mapping, etc.
- As of version 1.4, Java includes a new I/O system with improved performance and functionality
- The new I/O system is named “New I/O” (NIO) and is located under the *java.nio* package

“New” I/O vs. “Old” I/O

- The speed of the “new” I/O comes from using structures that are closer to the operating system’s way of performing I/O
  - That is, NIO employs I/O functionalities offered by most modern operating systems
- However, most of the “old” I/O classes have been re-implemented using NIO
  - So the “new old” I/O (post JDK 1.4) is much more efficient and scalable than the “old old” I/O (pre JDK 1.4)
- Still, the “new” I/O offers some functionalities not available in the “old” I/O, such as non-blocking operations
- From Java 8, NIO has operations that return streaming API streams, for example:
  - Files.lines(path) – stream of strings representing file lines
  - Files.list(path) – stream of file entries in a directory
  - Files.walk(path) – traverse the whole directory tree