Java Programming Tutorial
Basic Input & Output (I/O)

Programming simple I/O operations is easy, which involves only a few classes and methods. You could do it by looking at a few samples. Programming efficient, portable I/O is *extremely* difficult, especially if you have to deal with different character sets. This explains why there are so many I/O packages (nine in JDK 1.7)!

JDK has two sets of I/O packages:

1. the Standard I/O (in package `java.io`), introduced since JDK 1.0 for stream-based I/O, and
2. the New I/O (in packages `java.nio`), introduced in JDK 1.4, for more efficient buffer-based I/O.

JDK 1.5 introduces the *formatted text-I/O* via new classes `java.util.Scanner` and `Formatter`, and C-like `printf()` and `format()` methods for formatted output using format specifiers.

JDK 1.7 enhances supports for file I/O via the so-called NIO.2 (non-blocking I/O) in new package `java.nio.file` and its auxiliary packages. It also introduces a new `try`-with-resources syntax to simplify the coding of `close()` method.

1. File and Directory

1.1 Class java.io.File (Pre-JDK 7)

The class `java.io.File` can represent either a *file* or a *directory*. [JDK 1.7 introduces a more versatile `java.nio.file.Path`, which overcomes many limitations of `java.io.File`.]

A *path string* is used to locate a *file* or a *directory*. Unfortunately, path strings are system dependent, e.g., "`c:\myproject\java\Hello.java`" in Windows or "`/myproject/java/Hello.java`" in Unix/Mac.

- Windows use back-slash `'\'` as the *directory separator*; while Unixes/Mac use forward-slash `'/'`.
- Windows use semi-colon `';'` as *path separator* to separate a list of paths; while Unixes/Mac use colon `':'`.
- Windows use "`\r\n`" as *line delimiter* for text file; while Unixes use "`\n`" and Mac uses "`\r`".
- The "`c:\`" or "`\`" is called the *root*. Windows supports multiple roots, each maps to a drive (e.g., "`c:\`", "`d:\`"). Unixes/Mac has a single root ("`\`").

A path could be *absolute* (beginning from the root) or *relative* (which is relative to a reference directory). Special notations "`.`" and "`..`" denote the current directory and the parent directory, respectively.

The `java.io.File` class maintains these system-dependent properties, for you to write programs that are portable:

- *Directory Separator*: in `static` fields `File.separator` (as `String`) and `File.separatorChar`. [They failed to follow the Java naming convention for constants adopted since JDK 1.2.] As mentioned, Windows use backslash `'\'`; while Unixes/Mac use forward slash `'/'`.
- *Path Separator*: in `static` fields `File.pathSeparator` (as `String`) and `File.pathSeparatorChar`. As mentioned, Windows use semi-colon `';'` to separate a list of paths; while Unixes/Mac use colon `':'`.

You can construct a `File` instance with a path string or URI, as follows. Take note that the physical file/directory may or may not exist. A file URL takes the form of `file://...`, e.g., `file:///d:/docs/programming/java/test.html`.

```
public File(String pathString)
public File(String parent, String child)
public File(File parent, String child)
// Constructs a File instance based on the given path string.
```

```
public File(URI uri)
// Constructs a File instance by converting from the given file-URI "file://...."
```

For examples,

```
File file = new File("in.txt");     // A file relative to the current working directory
File file = new File("d:\\myproject\\java\\Hello.java");  // A file with absolute path
File dir  = new File("c:\\temp");   // A directory
```

For applications that you intend to distribute as JAR files, you should use the `URL` class to reference the resources, as it can reference disk files as well as JAR'ed files , for example,

```
java.net.URL url = this.getClass().getResource("icon.png");
```

Verifying Properties of a File/Directory

```
public boolean exists()       // Tests if this file/directory exists.
public long length()          // Returns the length of this file.
public boolean isDirectory()  // Tests if this instance is a directory.
public boolean isFile()       // Tests if this instance is a file.
public boolean canRead()      // Tests if this file is readable.
public boolean canWrite()     // Tests if this file is writable.
public boolean delete()       // Deletes this file/directory.
public void deleteOnExit()    // Deletes this file/directory when the program terminates.
public boolean renameTo(File dest) // Renames this file.
public boolean mkdir()        // Makes (Creates) this directory.
```

List Directory

For a directory, you can use the following methods to list its contents:

```
public String[] list()     // List the contents of this directory in a String-array
public File[] listFiles()  // List the contents of this directory in a File-array
```

Example: The following program recursively lists the contents of a given directory (similar to Unix's `"ls -r"` command).

```
// Recursively list the contents of a directory (Unix's "ls -r" command).
import java.io.File;
public class ListDirectoryRecusive {
   public static void main(String[] args) {
      File dir = new File("d:\\myproject\\test");  // Escape sequence needed for '\'
      listRecursive(dir);
   }

   public static void listRecursive(File dir) {
      if (dir.isDirectory()) {
         File[] items = dir.listFiles();
         for (File item : items) {
            System.out.println(item.getAbsoluteFile());
            if (item.isDirectory()) listRecursive(item);  // Recursive call
         }
      }
   }
}
```

(Advanced) List Directory with Filter

You can apply a filter to `list()` and `listFiles()`, to list only files that meet a certain criteria.

```
public String[] list(FilenameFilter filter)
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
```

The `interface java.io.FilenameFilter` declares one `abstract` method:

```
public boolean accept(File dirName, String fileName)
```

The `list()` and `listFiles()` methods does a *call-back* to `accept()` for each of the file/sub-directory produced. You can program your filtering criteria in `accept()`. Those files/sub-directories that result in a `false` return will be excluded.

Example: The following program lists only files that meet a certain filtering criteria.

```
// List files that end with ".java"
import java.io.File;
import java.io.FilenameFilter;
public class ListDirectoryWithFilter {
   public static void main(String[] args) {
      File dir = new File(".");   // current working directory
      if (dir.isDirectory()) {
```

```
         // List only files that meet the filtering criteria
         //  programmed in accept() method of FilenameFilter.
         String[] files = dir.list(new FilenameFilter() {
            public boolean accept(File dir, String file) {
               return file.endsWith(".java");
            }
         });  // an anonymous inner class as FilenameFilter
         for (String file : files) {
            System.out.println(file);
         }
      }
   }
}
```

1.2 java.io.File Class

Read "File I/O in JDK 1.7".

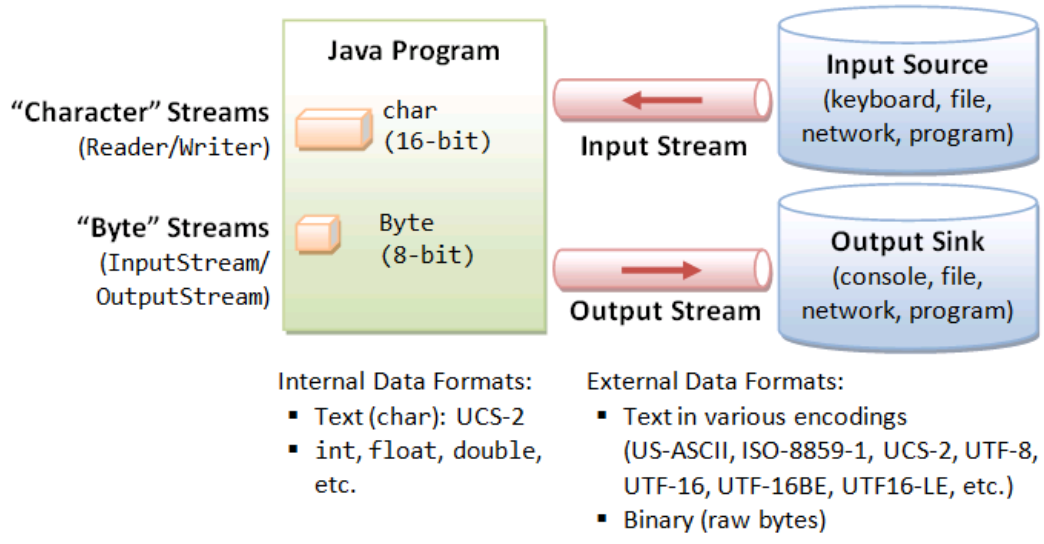
2. Stream I/O

Programs read inputs from data sources (e.g., keyboard, file, network, memory buffer, or another program) and write outputs to data sinks (e.g., display console, file, network, memory buffer, or another program). In Java standard I/O, inputs and outputs are handled by the so-called *streams*. A *stream* is a sequential and contiguous one-way flow of data (just like water or oil flows through the pipe). It is important to mention that Java does not differentiate between the various types of data sources or sinks (e.g., file or network) in stream I/O. They are all treated as a sequential flow of data. Input and output streams can be established from/to any data source/sink, such as files, network, keyboard/console or another program. The Java program receives data from a source by opening an input stream, and sends data to a sink by opening an output stream. All Java I/O streams are one-way (except the `RandomAccessFile`, which will be discussed later). If your program needs to perform both input and output, you have to open two streams - an input stream and an output stream.

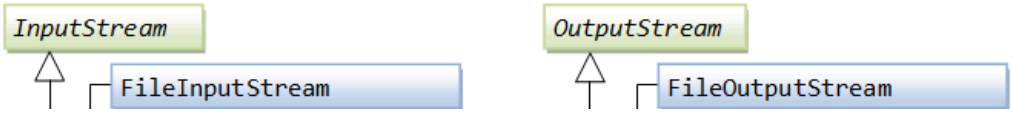
Stream I/O operations involve three steps:

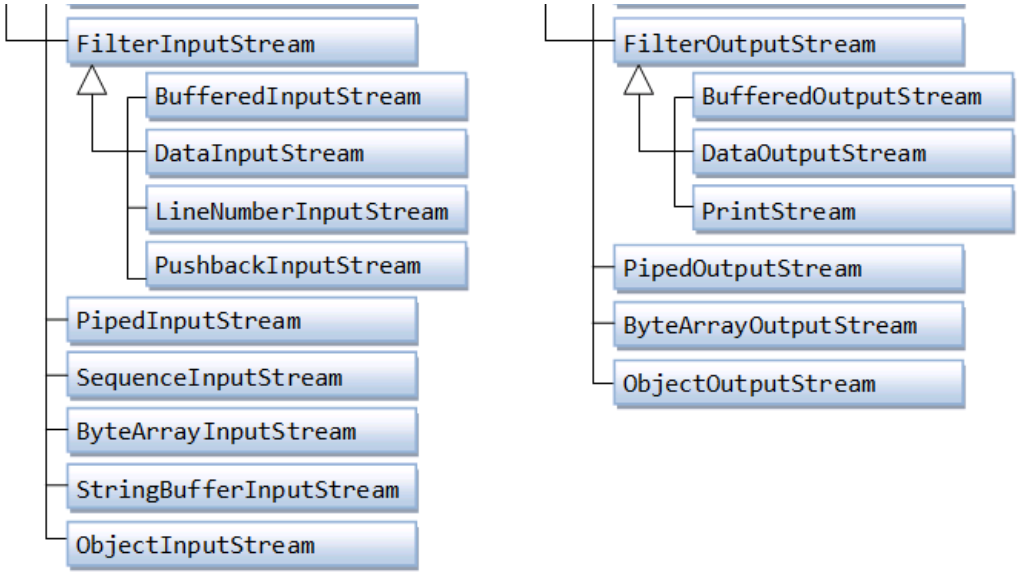
1. *Open* an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
2. *Read* from the opened input stream until "end-of-stream" encountered, or *write* to the opened output stream (and optionally flush the buffered output).
3. *Close* the input/output stream.

Java's I/O operations is more complicated than C/C++ to support internationalization (i18n). Java internally stores characters (`char` type) in 16-bit UCS-2 character set. But the external data source/sink could store characters in other character set (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, and many others), in fixed length of 8-bit or 16-bit, or in variable length of 1 to 4 bytes. [Read "Character Sets and Encoding Schemes"]. As a consequence, Java needs to differentiate between byte-based I/O for processing *raw bytes* or *binary data*, and character-based I/O for processing *texts* made up of characters.



3. Byte-Based I/O & Byte Streams





Byte streams are used to read/write *raw bytes* serially from/to an external device. All the byte streams are derived from the `abstract` superclasses `InputStream` and `OutputStream`, as illustrated in the class diagram.

3.1 Reading Input from InputStream

The `abstract` superclass `InputStream` declares an `abstract` method `read()` to read one data-byte from the input source:

```
public abstract int read() throws IOException
```

The `read()` method:

- returns the input byte read as an `int` in the range of 0 to 255, or
- returns -1 if "end of stream" condition is detected, or
- throws an `IOException` if it encounters an I/O error.

The `read()` method returns an `int` instead of a `byte`, because it uses -1 to indicate end-of-stream.

The `read()` method *blocks* until a byte is available, an I/O error occurs, or the "end-of-stream" is detected. The term "*block*" means that the method (and the program) will be suspended. The program will resume only when the method returns.

Two variations of `read()` methods are implemented in the `InputStream` for reading a block of bytes into a byte-array. It returns the number of bytes read, or -1 if "end-of-stream" encounters.

```
public int read(byte[] bytes, int offset, int length) throws IOException
// Read "length" number of bytes, store in bytes array starting from offset of index.
public int read(byte[] bytes) throws IOException
// Same as read(bytes, 0, bytes.length)
```

3.2 Writing Output to OutputStream

Similar to the input counterpart, the `abstract` superclass `OutputStream` declares an abstract method `write()` to write a data-byte to the output sink. `write()` takes an `int`. The least-significant byte of the `int` argument is written out; the upper 3 bytes are discarded. It throws an `IOException` if I/O error occurs (e.g., output stream has been closed).

```
public void abstract void write(int unsignedByte) throws IOException
```

Similar to the `read()`, two variations of the `write()` method to write a block of bytes from a byte-array are implemented:

```
public void write(byte[] bytes, int offset, int length) throws IOException
// Write "length" number of bytes, from the bytes array starting from offset of index.
public void write(byte[] bytes) throws IOException
// Same as write(bytes, 0, bytes.length)
```

3.3 Opening & Closing I/O Streams

You *open* an I/O stream by constructing an instance of the stream. Both the `InputStream` and the `OutputStream` provides a `close()` method to close the stream, which performs the necessary clean-up operations to free up the system resources.

```
public void close() throws IOException  // close this Stream
```

It is a good practice to explicitly close the I/O stream, by running `close()` in the `finally` clause of `try-catch-finally` to free up the system resources immediately when the stream is no longer needed. This could prevent serious resource leaks. Unfortunately, the `close()` method also

throws a `IOException`, and needs to be enclosed in a nested `try-catch` statement, as follows. This makes the codes somehow ugly.

```
FileInputStream in = null;
......
try {
   in = new FileInputStream(...);  // Open stream
   ......
   ......
} catch (IOException ex) {
   ex.printStackTrace();
} finally {  // always close the I/O streams
   try {
      if (in != null) in.close();
   } catch (IOException ex) {
      ex.printStackTrace();
   }
}
```

JDK 1.7 introduces a new `try`-with-resources syntax, which automatically closes all the opened resources after `try` or `catch`, as follows. This produces much neater codes.

```
try (FileInputStream in = new FileInputStream(...)) {
   ......
   ......
} catch (IOException ex) {
   ex.printStackTrace();
}  // Automatically closes all opened resource in try (...).
```

3.4 Flush the Output Stream

In addition, the `OutputStream` provides a `flush()` method to flush the remaining bytes from the output buffer.

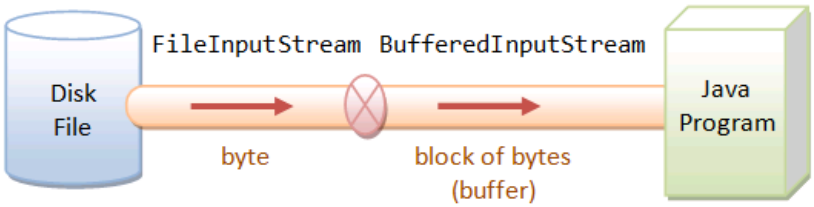
```
public void flush() throws IOException  // Flush the output
```

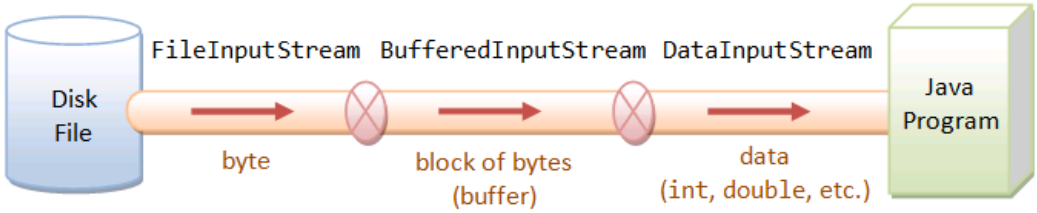
3.5 Implementation Subclasses of InputStream / OutputStream

`InputStream` and `OutputStream` are `abstract` classes that cannot be instantiated. You need to choose an appropriate concrete subclass to establish a connection to a physical device. For example, you can instantiate a `FileInputStream` or `FileOutputStream` to establish a stream to a physical disk file.

3.6 Layered (or Chained) I/O Streams

The I/O streams are often layered or chained with other I/O streams, for purposes such as buffering, filtering, or data-format conversion (between raw bytes and primitive types). For example, we can layer a `BufferedInputStream` to a `FileInputStream` for buffered input, and stack a `DataInputStream` in front for formatted data input (using primitives such as `int`, `double`), as illustrated in the following diagrams.





3.7 File I/O Byte-Oriented Stream: FileInputStream and FileOutputStream

`FileInputStream` and `FileOutputStream` are concrete implementations to the `abstract` classes `InputStream` and `OutputStream`, to support I/O from disk files.

3.8 Buffered I/O Byte-Oriented Stream: BufferedInputStream and BufferedOutputStream

The `read()`/`write()` method in `InputStream`/`OutputStream` are designed to read/write a single byte of data on each call. This is grossly inefficient, as each call is handled by the underlying operating system (which may trigger a disk access, or other expensive operations). *Buffering*, which reads/writes a block of bytes from the external device into/from a memory buffer in a single I/O operation, is commonly applied to speed up the I/O.

`FileInputStream`/`FileOutputStream` is not buffered. It is often chained to a `BufferedInputStream` or `BufferedOutputStream`, which provides the buffering. To chain the streams together, simply pass an instance of one stream into the constructor of another stream. For example, the following codes chain a `FileInputStream` to a `BufferedInputStream`, and finally, a `DataInputStream`:

```
FileInputStream fileIn = new FileInputStream("in.dat");
BufferedInputStream bufferIn = new BufferedInputStream(fileIn);
DataInputStream dataIn = new DataInputStream(bufferIn);
// or
DataInputStream in = new DataInputStream(
                        new BufferedInputStream(
                           new FileInputStream("in.dat")));
```

Example Copying a file byte-by-byte without Buffering.

```
import java.io.*;
public class FileCopyNoBuffer {  // Pre-JDK 7
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      FileInputStream in = null;
      FileOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      // Print file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      try {
         in = new FileInputStream(inFileStr);
         out = new FileOutputStream(outFileStr);

         startTime = System.nanoTime();
         int byteRead;
         // Read a raw byte, returns an int of 0 to 255.
         while ((byteRead = in.read()) != -1) {
            // Write the least-significant byte of int, drop the upper 3 bytes
            out.write(byteRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      } finally {  // always close the I/O streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

```
File size is 417455 bytes
Elapsed Time is 3781.500581 msec
```

This example copies a file by reading a byte from the input file and writing it to the output file. It uses `FileInputStream` and `FileOutputStream` directly without buffering. Notice that most the I/O methods "throws" `IOException`, which must be caught or declared to be thrown. The method `close()` is programmed inside the `finally` clause. It is guaranteed to be run after `try` or `catch`. However, method `close()` also throws an `IOException`, and therefore must be enclosed inside a nested `try-catch` block, which makes the codes a little ugly.

I used `System.nanoTime()`, which was introduced in JDK 1.5, for a more accurate measure of the elapsed time, instead of the legacy not-so-precise `System.currentTimeMillis()`. The output shows that it took about 4 seconds to copy a 400KB file.

As mentioned, JDK 1.7 introduces a new `try`-with-resources syntax, which automatically closes all the resources opened, after `try` or `catch`. For example, the above example can be re-written in a much neater manner as follow:

```
import java.io.*;
public class FileCopyNoBufferJDK7 {
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      long startTime, elapsedTime;  // for speed benchmarking
```

```
      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      // "try-with-resources" automatically closes all opened resources.
      try (FileInputStream  in  = new FileInputStream(inFileStr);
           FileOutputStream out = new FileOutputStream(outFileStr)) {

         startTime = System.nanoTime();
         int byteRead;
         // Read a raw byte, returns an int of 0 to 255.
         while ((byteRead = in.read()) != -1) {
            // Write the least-significant byte of int, drop the upper 3 bytes
            out.write(byteRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

Example 2: Copying a file with a Programmer-Managed Buffer.

```
import java.io.*;
public class FileCopyUserBuffer {  // Pre-JDK 7
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      FileInputStream in = null;
      FileOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      try {
         in = new FileInputStream(inFileStr);
         out = new FileOutputStream(outFileStr);
         startTime = System.nanoTime();
         byte[] byteBuf = new byte[4096];    // 4K byte-buffer
         int numBytesRead;
         while ((numBytesRead = in.read(byteBuf)) != -1) {
            out.write(byteBuf, 0, numBytesRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      } finally {  // always close the streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) { ex.printStackTrace(); }
      }
   }
}
```

```
File size is 417455 bytes
Elapsed Time is 2.938921 msec
```

This example again uses `FileInputStream` and `FileOutputStream` directly. However, instead of reading/writing one byte at a time, it reads/writes a 4KB block. This program took only 3 millisecond - a more than 1000 times speed-up compared with the previous example.

Larger buffer size, up to a certain limit, generally improves the I/O performance. However, there is a trade-off between speed-up the the memory usage. For file copying, a large buffer is certainly recommended. But for reading just a few bytes from a file, large buffer simply wastes the memory.

I re-write the program using JDK 1.7, and try on various buffer size on a much bigger file of 26MB.

```
import java.io.*;
public class FileCopyUserBufferLoopJDK7 {
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      long startTime, elapsedTime;  // for speed benchmarking

      // Check file length
```

```
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      int[] bufSizeKB = {1, 2, 4, 8, 16, 32, 64, 256, 1024};  // in KB
      int bufSize;  // in bytes

      for (int run = 0; run < bufSizeKB.length; ++run) {
         bufSize = bufSizeKB[run] * 1024;
         try (FileInputStream in = new FileInputStream(inFileStr);
              FileOutputStream out = new FileOutputStream(outFileStr)) {
            startTime = System.nanoTime();
            byte[] byteBuf = new byte[bufSize];
            int numBytesRead;
            while ((numBytesRead = in.read(byteBuf)) != -1) {
               out.write(byteBuf, 0, numBytesRead);
            }
            elapsedTime = System.nanoTime() - startTime;
            System.out.printf("%4dKB: %6.2fmsec%n", bufSizeKB[run], (elapsedTime / 1000000.0));
            //System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

```
File size is 26246026 bytes [26 MB]
   1KB: 573.54msec
   2KB: 316.43msec
   4KB: 178.47msec
   8KB: 116.32msec
  16KB:  85.61msec
  32KB:  65.92msec
  64KB:  57.81msec
 256KB:  63.38msec
1024KB:  98.87msec
```

Increasing buffer size helps only up to a certain point?!

Example Copying a file with Buffered Streams.

```
import java.io.*;
public class FileCopyBufferedStream {  // Pre-JDK 7
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      BufferedInputStream in = null;
      BufferedOutputStream out = null;
      long startTime, elapsedTime;  // for speed benchmarking

      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      try {
         in  = new BufferedInputStream(new FileInputStream(inFileStr));
         out = new BufferedOutputStream(new FileOutputStream(outFileStr));
         startTime = System.nanoTime();
         int byteRead;
         while ((byteRead = in.read()) != -1) {  // Read byte-by-byte from buffer
            out.write(byteRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      } finally {            // always close the streams
         try {
            if (in != null) in.close();
            if (out != null) out.close();
         } catch (IOException ex) { ex.printStackTrace(); }
      }
   }
}
```

```
File size is 417455 bytes
Elapsed Time is 61.834954 msec
```

In this example, I chain the `FileInputStream` with `BufferedInputStream`, `FileOutputStream` with `BufferedOutputStream`, and read/write byte-by-byte. The JRE decides on the buffer size. The program took 62 milliseconds, about 60 times speed-up compared with example 1, but slower

than the programmer-managed buffer.

The JDK 1.7 version of the above example is as follows:

```
import java.io.*;
public class FileCopyBufferedStreamJDK7 {
   public static void main(String[] args) {
      String inFileStr = "test-in.jpg";
      String outFileStr = "test-out.jpg";
      long startTime, elapsedTime;  // for speed benchmarking

      // Check file length
      File fileIn = new File(inFileStr);
      System.out.println("File size is " + fileIn.length() + " bytes");

      try (BufferedInputStream in = new BufferedInputStream(new FileInputStream(inFileStr));
           BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(outFileStr))) {
         startTime = System.nanoTime();
         int byteRead;
         while ((byteRead = in.read()) != -1) {
            out.write(byteRead);
         }
         elapsedTime = System.nanoTime() - startTime;
         System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

3.9 Formatted Data Input/Output Stream

The `DataInputStream` and `DataOutputStream` can be stacked on top of any `InputStream` and `OutputStream` to parse the raw bytes so as to perform I/O operations in the desired data format, such as `int` and `double`.

To use `DataInputStream` for formatted input, you can chain up the input streams as follows:

```
DataInputStream in = new DataInputStream(
                        new BufferedInputStream(
                           new FileInputStream("in.dat")));
```

`DataInputStream` implements `DataInput` interface, which provides methods to read formatted primitive data and `String`, such as:

```
// 8 Primitives
public final int readInt() throws IOExcpetion;       // Read 4 bytes and convert into int
public final double readDoube() throws IOExcpetion;  // Read 8 bytes and convert into double
public final byte readByte() throws IOExcpetion;
public final char readChar() throws IOExcpetion;
public final short readShort() throws IOExcpetion;
public final long readLong() throws IOExcpetion;
public final boolean readBoolean() throws IOExcpetion;    // Read 1 byte. Convert to false if zero
public final float readFloat() throws IOExcpetion;

public final int readUnsignedByte() throws IOExcpetion;   // Read 1 byte in [0, 255] upcast to int
public final int readUnsignedShort() throws IOExcpetion;  // Read 2 bytes in [0, 65535], same as char, upcast to int
public final void readFully(byte[] b, int off, int len) throws IOException;
public final void readFully(byte[] b) throws IOException;

// Strings
public final String readLine() throws IOException;
     // Read a line (until newline), convert each byte into a char - no unicode support.
public final String readUTF() throws IOException;
    // read a UTF-encoded string with firt two bytes indicating its UTF bytes length

public final int skipBytes(int n)  // Skip a number of bytes
```

Similarly, you can stack the `DataOutputStream` as follows:

```
DataOutputStream out = new DataOutputStream(
                          new BufferedOutputStream(
                             new FileOutputStream("out.dat")));
```

`DataOutputStream` implements `DataOutput` interface, which provides methods to write formatted primitive data and `String`. For examples,

```
// 8 primitive types
public final void writeInt(int i) throws IOExcpetion;      // Write the int as 4 bytes
public final void writeFloat(float f) throws IOExcpetion;
public final void writeDoube(double d) throws IOExcpetion; // Write the double as 8 bytes
public final void writeByte(int b) throws IOExcpetion;     // least-significant byte
```

```
public final void writeShort(int s) throws IOExcpetion;    // two lower bytes
public final void writeLong(long l) throws IOExcpetion;
public final void writeBoolean(boolean b) throws IOExcpetion;
public final void writeChar(int i) throws IOExcpetion;

// String
public final void writeBytes(String str) throws IOExcpetion;
     // least-significant byte of each char
public final void writeChars(String str) throws IOExcpetion;
     // Write String as UCS-2 16-bit char, Big-endian (big byte first)
public final void writeUTF(String str) throws IOException;
     // Write String as UTF, with first two bytes indicating UTF bytes length

public final void write(byte[] b, int off, int len) throws IOException
public final void write(byte[] b) throws IOException
public final void write(int b) throws IOException     // Write the least-significant byte
```

Example: The following program writes some primitives to a disk file. It then reads the raw bytes to check how the primitives were stored. Finally, it reads the data as primitives.

```
import java.io.*;
public class TestDataIOStream {
   public static void main(String[] args) {
      String filename = "data-out.dat";
      String message = "Hi,您好!";

      // Write primitives to an output file
      try (DataOutputStream out =
              new DataOutputStream(
                 new BufferedOutputStream(
                    new FileOutputStream(filename)))) {
         out.writeByte(127);
         out.writeShort(0xFFFF);  // -1
         out.writeInt(0xABCD);
         out.writeLong(0x1234_5678);  // JDK 7 syntax
         out.writeFloat(11.22f);
         out.writeDouble(55.66);
         out.writeBoolean(true);
         out.writeBoolean(false);
         for (int i = 0; i < message.length(); ++i) {
            out.writeChar(message.charAt(i));
         }
         out.writeChars(message);
         out.writeBytes(message);
         out.flush();
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read raw bytes and print in Hex
      try (BufferedInputStream in =
              new BufferedInputStream(
                 new FileInputStream(filename))) {
         int inByte;
         while ((inByte = in.read()) != -1) {
            System.out.printf("%02X ", inByte);   // Print Hex codes
         }
         System.out.printf("%n%n");
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read primitives
      try (DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(
                    new FileInputStream(filename)))) {
         System.out.println("byte:    " + in.readByte());
         System.out.println("short:   " + in.readShort());
         System.out.println("int:     " + in.readInt());
         System.out.println("long:    " + in.readLong());
         System.out.println("float:   " + in.readFloat());
         System.out.println("double:  " + in.readDouble());
         System.out.println("boolean: " + in.readBoolean());
         System.out.println("boolean: " + in.readBoolean());

         System.out.print("char:    ");
         for (int i = 0; i < message.length(); ++i) {
            System.out.print(in.readChar());
         }
```

```
         System.out.println();

         System.out.print("chars:   ");
         for (int i = 0; i < message.length(); ++i) {
            System.out.print(in.readChar());
         }
         System.out.println();

         System.out.print("bytes:   ");
         for (int i = 0; i < message.length(); ++i) {
            System.out.print((char)in.readByte());
         }
         System.out.println();
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

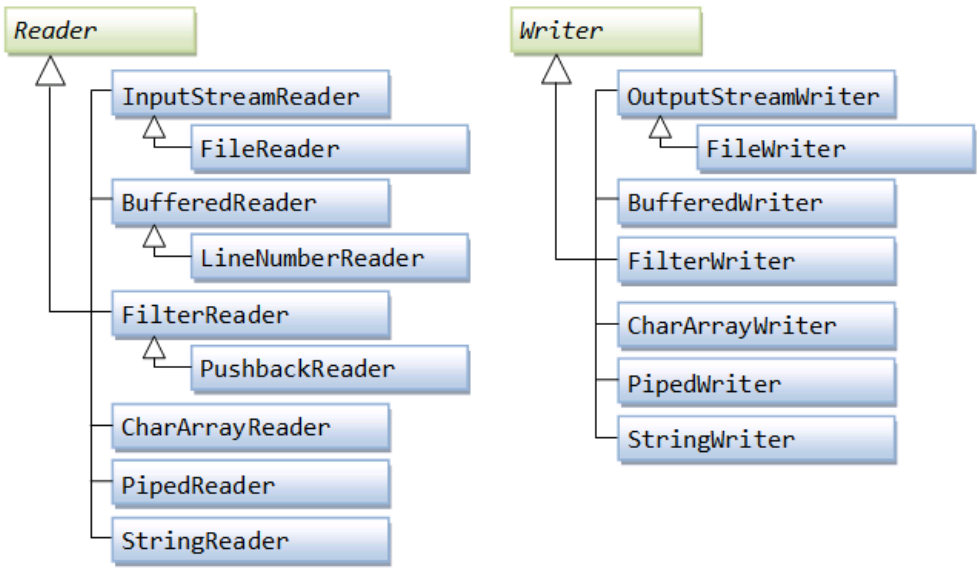
```
7F   FF FF 00 00 AB CD 00 00 00 00 00 0F 42 3F
byte short int         long
41 33 85 1F 40 4B D4 7A E1 47 AE 14
float       double
01      00
boolean boolean
00 48 00 69 00 2C 60 A8 59 7D 00 21
H     i     ,     您    好    !
00 48 00 69 00 2C 60 A8 59 7D 00 21
H     i     ,     您    好    !
48 69 2C A8 7D 21
[low byte of the char only]

byte:    127
short:   -1
int:     43981
long:    305419896
float:   11.22
double:  55.66
boolean: true
boolean: false
char:    Hi,您好!
chars:   Hi,您好!
bytes:   Hi,?}!
```

The data stored in the disk are exactly in the same form as in the Java program internally (e.g., UCS-2 for characters). The byte-order is big-endian (big byte first, or most significant byte in lowest address).

3.10 Network I/O

[In Java Networking]

4. Character-Based I/O & Character Strea



Java internally stores characters (`char` type) in 16-bit UCS-2 character set. But the external data source/sink could store characters in other character set (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, and many others), in fixed length of 8-bit or 16-bit, or in variable length of 1 to 4 bytes. [Read "Character Sets and Encoding Schemes"]. Hence, Java has to differentiate between byte-based I/O for processing 8-bit raw bytes, and character-based I/O for processing texts. The character streams needs to *translate* between the character set used by external I/O devices and Java internal UCS-2 format. For example, the character '您' is stored as "`60 A8`" in UCS-2 (Java internal), "`E6 82 A8`" in UTF8, "`C4 FA`" in GBK/GB2312, and "`B1 7A`" in BIG5. If this character is to be written to a file uses UTF-8, the character stream needs to translate "`60 A8`" to "`E6 82 A8`". The reserve takes place in a reading operation.

The byte/character streams refer to the unit of operation within the Java programs, which does not necessary correspond to the amount of data transferred from/to the external I/O devices. This is because some charsets use fixed-length of 8-bit (e.g., US-ASCII, ISO-8859-1) or 16-bit (e.g., UCS-16), while some use variable-length of 1-4 bytes (e.g., UTF-8, UTF-16, UTF-16-BE, UTF-16-LE, GBK, BIG5). When a character stream is used to read an 8-bit ASCII file, an 8-bit data is read from the file and put into the 16-bit `char` location of the Java program.

4.1 Abstract Superclasses and Interfaces

Other than the unit of operation and charset conversion (which is extremely complex), character-based I/O is almost identical to byte-based I/O. Instead of `InputStream` and `OutputStream`, we use `Reader` and `Writer` for character-based I/O.

The `abstract` superclass `Reader` operates on `char`. It declares an `abstract` method `read()` to read one character from the input source. `read()` returns the character as an `int` between 0 to 65535 (a `char` in Java can be treated as an unsigned 16-bit integer); or -1 if end-of-stream is detected; or throws an `IOException` if I/O error occurs. There are also two variations of `read()` to read a block of characters into `char`-array.

```
public abstract int read() throws IOException
public int read(char[] chars, int offset, int length) throws IOException
public int read(char[] chars) throws IOException
```

The `abstract` superclass `Writer` declares an `abstract` method `write()`, which writes a character to the output sink. The lower 2 bytes of the `int` argument is written out; while the upper 2 bytes are discarded.

```
public void abstract void write(int aChar) throws IOException
public void write(char[] chars, int offset, int length) throws IOException
public void write(char[] chars) throws IOException
```

4.2 File I/O Character Streams: FileReader and FileWriter

`FileReader` and `FileWriter` are concrete implementations to the `abstract` superclasses `Reader` and `Writer`, to support I/O from disk files. `FileReader/FileWriter` assumes that the *default character encoding* (*charset*) is used for the disk file. The default charset is kept in the JVM's system property "`file.encoding`". You can get the default charset via `static` method `java.nio.charset.Charset.defaultCharset()` or `System.getProperty("file.encoding")`. It is probable safe to use `FileReader/FileWriter` for ASCII texts, provided that the default charset is compatible to ASCII (such as US-ASCII, ISO-8859-x, UTF-8, and many others, but NOT UTF-16, UTF-16BE, UTF-16LE and many others). Use of `FileReader/FileWriter` is NOT recommended as you have no control of the file encoding charset.

4.3 Buffered I/O Character Streams: BufferedReader and BufferedWriter

`BufferedReader` and `BufferedWriter` can be stacked on top of `FileReader/FileWriter` or other character streams to perform buffered I/O, instead of character-by-character. `BufferedReader` provides a new method `readLine()`, which reads a line and returns a `String` (without the line delimiter). Lines could be delimited by "`\n`" (Unix), "`\r\n`" (Windows), or "`\r`" (Mac).

Example

```
import java.io.*;
// Write a text message to an output file, then read it back.
// FileReader/FileWriter uses the default charset for file encoding.
public class BufferedFileReaderWriterJDK7 {
   public static void main(String[] args) {
      String strFilename = "out.txt";
      String message = "Hello, world!\nHello, world again!\n";  // 2 lines of texts

      // Print the default charset
      System.out.println(java.nio.charset.Charset.defaultCharset());

      try (BufferedWriter out = new BufferedWriter(new FileWriter(strFilename))) {
         out.write(message);
         out.flush();
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      try (BufferedReader in = new BufferedReader(new FileReader(strFilename))) {
         String inLine;
         while ((inLine = in.readLine()) != null) {  // exclude newline
            System.out.println(inLine);
```

```
         }
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

4.4 Character Set (java.nio.charset package)

JDK 1.4 provides a new package `java.nio.charset` as part of NIO (New IO) to support character translation between the Unicode (UCS-2) used internally in Java program and external devices which could be encoded in any other format (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, UTF-16BE, UTF-16LE, and etc.)

The main class `java.nio.charset.Charset` provides `static` methods for testing whether a particular charset is supported, locating charset instances by name, and listing all the available charsets and the default charset.

```
public static SortedMap<String,Charset> availableCharsets()  // lists all the available charsets
public static Charset defaultCharset()                // Returns the default charset
public static Charset forName(String charsetName)     // Returns a Charset instance for the given charset name (in String)
public static boolean isSupported(String charsetName)  // Tests if this charset name is supported
```

Example

```
import java.nio.charset.Charset;
public class TestCharset {
   public static void main(String[] args) {
      // Print the default Charset
      System.out.println("The defalut charset is " + Charset.defaultCharset());
      System.out.println("The defalut charset is " + System.getProperty("file.encoding"));

      // Print the list of available Charsets in name=Charset
      System.out.println("The available charsets are:");
      System.out.println(Charset.availableCharsets());

      // Check if the given charset name is supported
      System.out.println(Charset.isSupported("UTF-8"));  // true
      System.out.println(Charset.isSupported("UTF8"));   // true
      System.out.println(Charset.isSupported("UTF_8"));  // false

      // Get an instance of a Charset
      Charset charset = Charset.forName("UTF8");
      // Print this Charset name
      System.out.println(charset.name());     // "UTF-8"
      // Print all the other aliases
      System.out.println(charset.aliases());  // [UTF8, unicolor-1-1-utf-8]
   }
}
```

The default charset for file encoding is kept in the system property "`file.encoding`". To change the JVM's default charset for file encoding, you can use command-line VM option "`-Dfile.encoding`". For example, the following command run the program with default charset of UTF-8.

```
> java -Dfile.encoding=UTF-8 TestCharset
```

Most importantly, the `Charset` class provides methods to encode/decode characters from UCS-2 used in Java program and the specific charset used in the external devices (such as UTF-8).

```
public final ByteBuffer encode(String s)
public final ByteBuffer encode(CharBuffer cb)
// Encodes Unicode UCS-2 characters in the CharBuffer/String
//  into a "byte sequence" using this charset, and returns a ByteBuffer.

public final CharBuffer decode(ByteBuffer bb)
// Decode the byte sequence encoded using this charset in the ByteBuffer
//  to Unicode UCS-2, and return a charBuffer.
```

The `encode()`/`decode()` methods operate on `ByteBuffer` and `CharBuffer` introduced also in JDK 1.4, which will be explain in the New I/O section.

Example The following example encodes some Unicode texts in various encoding scheme, and display the Hex codes of the encoded byte sequences.

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;

public class TestCharsetEncodeDecode {
   public static void main(String[] args) {
```

```
      // Try these charsets for encoding
      String[] charsetNames = {"US-ASCII", "ISO-8859-1", "UTF-8", "UTF-16",
                               "UTF-16BE", "UTF-16LE", "GBK", "BIG5"};

      String message = "Hi,您好!";  // Unicode message to be encoded
      // Print UCS-2 in hex codes
      System.out.printf("%10s: ", "UCS-2");
      for (int i = 0; i < message.length(); ++i) {
         System.out.printf("%04X ", (int)message.charAt(i));
      }
      System.out.println();

      for (String charsetName: charsetNames) {
         // Get a Charset instance given the charset name string
         Charset charset = Charset.forName(charsetName);
         System.out.printf("%10s: ", charset.name());

         // Encode the Unicode UCS-2 characters into a byte sequence in this charset.
         ByteBuffer bb = charset.encode(message);
         while (bb.hasRemaining()) {
            System.out.printf("%02X ", bb.get());  // Print hex code
         }
         System.out.println();
         bb.rewind();
      }
   }
}
```

```
     UCS-2: 0048 0069 002C 60A8 597D 0021 [16-bit fixed-length]
            H    i    ,    您   好   !
  US-ASCII: 48 69 2C 3F 3F 21 [8-bit fixed-length]
            H  i  ,  ?  ?  !
ISO-8859-1: 48 69 2C 3F 3F 21 [8-bit fixed-length]
            H  i  ,  ?  ?  !
     UTF-8: 48 69 2C E6 82 A8 E5 A5 BD 21 [1-4 bytes variable-length]
            H  i  ,  您       好       !
    UTF-16: FE FF 00 48 00 69 00 2C 60 A8 59 7D 00 21 [2-4 bytes variable-length]
            BOM   H     i     ,     您    好    !     [Byte-Order-Mark indicates Big-Endian]
  UTF-16BE: 00 48 00 69 00 2C 60 A8 59 7D 00 21 [2-4 bytes variable-length]
            H     i     ,     您    好    !
  UTF-16LE: 48 00 69 00 2C 00 A8 60 7D 59 21 00 [2-4 bytes variable-length]
            H     i     ,     您    好    !
       GBK: 48 69 2C C4 FA BA C3 21 [1-2 bytes variable-length]
            H  i  ,  您    好    !
      Big5: 48 69 2C B1 7A A6 6E 21 [1-2 bytes variable-length]
            H  i  ,  您    好    !
```

Example The following example tries out the encoding/decoding on `CharBuffer` and `ByteBuffer`. Buffers will be discussed later under New I/O.

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.Charset;

public class TestCharsetEncodeByteBuffer {
   public static void main(String[] args) {
      byte[] bytes = {0x00, 0x48, 0x00, 0x69, 0x00, 0x2C,
                      0x60, (byte)0xA8, 0x59, 0x7D, 0x00, 0x21};  // "Hi,您好!"
      // Print UCS-2 in hex codes
      System.out.printf("%10s: ", "UCS-2");
      for (int i = 0; i < bytes.length; ++i) {
         System.out.printf("%02X ", bytes[i]);
      }
      System.out.println();

      Charset charset = Charset.forName("UTF-8");
      // Encode from UCS-2 to UTF-8
      // Create a ByteBuffer by wrapping a byte array
      ByteBuffer bb = ByteBuffer.wrap(bytes);
      // Create a CharBuffer from a view of this ByteBuffer
      CharBuffer cb = bb.asCharBuffer();
      ByteBuffer bbOut = charset.encode(cb);
      // Print hex code
      System.out.printf("%10s: ", charset.name());
      while (bbOut.hasRemaining()) {
         System.out.printf("%02X ", bbOut.get());
      }
      System.out.println();

      // Decode from UTF-8 to UCS-2
      bbOut.rewind();
```

```
      CharBuffer cbOut = charset.decode(bbOut);
      System.out.printf("%10s: ", "UCS-2");
      while (cbOut.hasRemaining()) {
         char aChar = cbOut.get();
         System.out.printf("'%c'[%04X] ", aChar, (int)aChar);  // Print char & hex code
      }
      System.out.println();
   }
}
```

```
UCS-2: 00 48 00 69 00 2C 60 A8 59 7D 00 21
UTF-8: 48 69 2C E6 82 A8 E5 A5 BD 21
UCS-2: 'H'[0048] 'i'[0069] ','[002C] '您'[60A8] '好'[597D] '!'[0021]
```

4.5 Text File I/O Using InputStreamReader and OutputStreamWriter

As mentioned, Java internally stores characters (`char` type) in 16-bit UCS-2 character set. But the external data source/sink could store characters in other character set (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, and many others), in fixed length of 8-bit or 16-bit, or in variable length of 1 to 4 bytes. The `FileReader`/`FileWriter` introduced earlier uses the default charset for decoding/encoding, resulted in non-portable programs.

To choose the charset, you need to use `InputStreamReader` and `OutputStreamWriter`. `InputStreamReader` and `OutputStreamWriter` are considered to be byte-to-character "bridge" streams.

You can choose the character set in the `InputStreamReader`'s constructor:

```
public InputStreamReader(InputStream in)   // Use default charset
public InputStreamReader(InputStream in, String charsetName) throws UnsupportedEncodingException
public InputStreamReader(InputStream in, Charset cs)
```

You can list the available charsets via `static` method `java.nio.charset.Charset.availableCharsets()`. The commonly-used `Charset` names supported by Java are:

- "`US-ASCII`": 7-bit ASCII (aka ISO646-US)
- "`ISO-8859-1`": Latin-1
- "`UTF-8`": Most commonly-used encoding scheme for Unicode
- "`UTF-16BE`": Big-endian (big byte first) (big-endian is usually the default)
- "`UTF-16LE`": Little-endian (little byte first)
- "`UTF-16`": with a 2-byte BOM (Byte-Order-Mark) to specify the byte order. FE FF indicates big-endian, FF FE indicates little-endian.

As the `InputStreamReader`/`OutputStreamWriter` often needs to read/write in multiple bytes, it is best to wrap it with a `BufferedReader`/`BufferedWriter`.

Example The following program writes Unicode texts to a disk file using various charsets for file encoding. It then reads the file byte-by-byte (via a byte-based input stream) to check the encoded characters in the various charsets. Finally, it reads the file using the character-based reader.

```
import java.io.*;
// Write texts to file using OutputStreamWriter specifying its charset encoding.
// Read byte-by-byte using FileInputStream.
// Read char-by-char using InputStreamReader specifying its charset encoding.
public class TextFileEncodingJDK7 {
   public static void main(String[] args) {
      String message = "Hi,您好!";   // with non-ASCII chars
      // Java internally stores char in UCS-2/UTF-16
      // Print the characters stored with Hex codes
      for (int i = 0; i < message.length(); ++i) {
         char aChar = message.charAt(i);
         System.out.printf("[%d]'%c'(%04X) ", (i+1), aChar, (int)aChar);
      }
      System.out.println();

      // Try these charsets for encoding text file
      String[] csStrs = {"UTF-8", "UTF-16BE", "UTF-16LE", "UTF-16", "GB2312", "GBK", "BIG5"};
      String outFileExt = "-out.txt";   // Output filenames are "charset-out.txt"

      // Write text file in the specified file encoding charset
      for (int i = 0; i < csStrs.length; ++i) {
         try (OutputStreamWriter out =
                 new OutputStreamWriter(
                    new FileOutputStream(csStrs[i] + outFileExt), csStrs[i]);
              BufferedWriter bufOut = new BufferedWriter(out)) {  // Buffered for efficiency
            System.out.println(out.getEncoding());  // Print file encoding charset
            bufOut.write(message);
            bufOut.flush();
         } catch (IOException ex) {
            ex.printStackTrace();
```

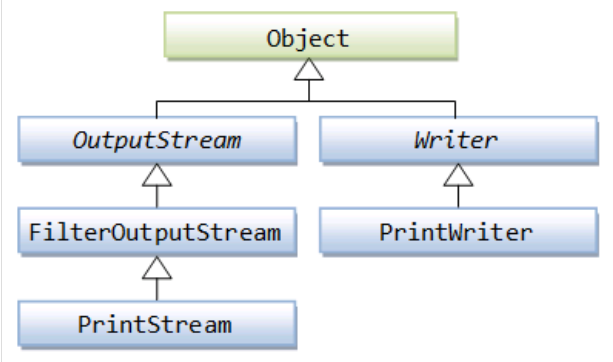
```
         }
      }

      // Read raw bytes from various encoded files
      //   to check how the characters were encoded.
      for (int i = 0; i < csStrs.length; ++i) {
         try (BufferedInputStream in = new BufferedInputStream(  // Buffered for efficiency
                 new FileInputStream(csStrs[i] + outFileExt))) {
            System.out.printf("%10s", csStrs[i]);    // Print file encoding charset
            int inByte;
            while ((inByte = in.read()) != -1) {
               System.out.printf("%02X ", inByte);   // Print Hex codes
            }
            System.out.println();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }

      // Read text file with character-stream specifying its encoding.
      // The char will be translated from its file encoding charset to
      //   Java internal UCS-2.
      for (int i = 0; i < csStrs.length; ++i) {
         try (InputStreamReader in =
                 new InputStreamReader(
                    new FileInputStream(csStrs[i] + outFileExt), csStrs[i]);
              BufferedReader bufIn = new BufferedReader(in)) {  // Buffered for efficiency
            System.out.println(in.getEncoding());  // print file encoding charset
            int inChar;
            int count = 0;
            while ((inChar = in.read()) != -1) {
               ++count;
               System.out.printf("[%d]'%c'(%04X) ", count, (char)inChar, inChar);
            }
         System.out.println();
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }
}
```

```
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

UTF-8:    48 69 2C E6 82 A8 E5 A5 BD 21
          H  i  ,  您       好       !
UTF-16BE: 00 48 00 69 00 2C 60 A8 59 7D 00 21
          H     i     ,     您    好    !
UTF-16LE: 48 00 69 00 2C 00 A8 60 7D 59 21 00
          H     i     ,     您    好    !
UTF-16:   FE FF 00 48 00 69 00 2C 60 A8 59 7D 00 21
          BOM   H     i     ,     您    好    !
GB2312:   48 69 2C C4 FA BA C3 21
          H  i  ,  您    好    !
GBK:      48 69 2C C4 FA BA C3 21
          H  i  ,  您    好    !
BIG5:     48 69 2C B1 7A A6 6E 21
          H  i  ,  您    好    !

UTF8
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

UnicodeBigUnmarked [UTF-16BE without BOM]
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

UnicodeLittleUnmarked [UFT-16LE without BOM]
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

UTF-16
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

EUC_CN [GB2312]
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

GBK
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

Big5
[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)
```

As seen from the output, the characters 您好 is encoded differently in different charsets. Nonetheless, the `InputStreamReader` is able to translate the characters into the same UCS-2 used in Java program.

5. java.io.PrintStream vs. java.io.PrintWriter

The byte-based `java.io.printSteam` supports convenient printing methods such as `print()` and `println()` for printing primitives and text string. Primitives are converted to their string representation for printing. The `printf()` and `format()` were introduced in JDK 1.5 for formatting output with former specifiers. `printf()` and `format()` are identical.



A `PrintStream` never throws an `IOException`. Instead, it sets an internal flag which can be checked via the `checkError()` method. A `PrintStream` can also be created to flush the output automatically. That is, the `flush()` method is automatically invoked after a byte array is written, one of the `println()` methods is invoked, or after a newline (`'\n'`) is written.

The standard output and error streams (`System.out` and `System.err`) belong to `PrintStream`.

All characters printed by a `PrintStream` are converted into bytes using the default character encoding. The `PrintWriter` class should be used in situations that require writing characters rather than bytes.

The character-stream `PrintWriter` is similar to `PrintStream`, except that it write in characters instead of bytes. The `PrintWriter` also supports all the convenient printing methods `print()`, `println()`, `printf()` and `format()`. It never throws an `IOException` and can optionally be created to support automatic flushing.

[TODO] Example to show the difference between `PrintStream` and `PrintWriter`.

6. Object Serialization and Object Stream

Data streams (`DataInputStream` and `DataOutputStream`) allow you to read and write primitive data (such as `int`, `double`) and `String`, rather than individual bytes. Object streams (`ObjectInputStream` and `ObjectOutputStream`) go one step further to allow you to read and write entire objects (such as `Date`, `ArrayList` or any custom objects).

Object serialization is the process of representing a "particular state of an object" in a serialized bit-stream, so that the bit stream can be written out to an external device (such as a disk file or network). The bit-stream can later be re-constructed to recover the state of that object. Object serialization is necessary to save a state of an object into a disk file for *persistence* or sent the object across the network for applications such as Web Services, Distributed-object applications, and Remote Method Invocation (RMI).

In Java, object that requires to be serialized must implement `java.io.Serializable` or `java.io.Externalizable` interface. `Serializable` interface is an *empty* interface (or *tagged* interface) with nothing declared. Its purpose is simply to declare that particular object is serializable.

6.1 ObjectInputStream and ObjectOutputStream

The `ObjectInputStream` and `ObjectOutputStream` can be used to serialize an object into a bit-stream and transfer it to/from an I/O streams, via these methods:

```
public final Object readObject() throws IOException, ClassNotFoundException;
public final void writeObject(Object obj) throws IOException;
```

`ObjectInputStream` and `ObjectOutputStream` must be stacked on top of a concrete implementation of `InputStream` or `OutputStream`, such as `FileInputStream` or `FileOutputStream`.

For example, the following code segment writes objects to a disk file. The "`.ser`" is the convention for serialized object file type.

```
ObjectOutputStream out =
   new ObjectOutputStream(
      new BufferedOutputStream(
         new FileOutputStream("object.ser")));
out.writeObject("The current Date and Time is "); // write a String object
out.writeObject(new Date());                      // write a Date object
out.flush();
out.close();
```

To read and re-construct the object back in a program, use the method `readObject()`, which returns an `java.lang.Object`. Downcast the `Object` back to its original type.

```
ObjectInputStream in =
   new ObjectInputStream(
      new BufferedInputStream(
```

```
         new FileInputStream("object.ser")));
String str = (String)in.readObject();
Date d = (Date)in.readObject(new Date());  // downcast
in.close();
```

Example Object serialization

```
import java.io.*;

public class ObjectSerializationTest {
   public static void main(String[] args) {
      String filename = "object.ser";
      int numObjs = 5;

      // Write objects
      try (ObjectOutputStream out =
              new ObjectOutputStream(
                 new BufferedOutputStream(
                    new FileOutputStream(filename)))) {

         // Create an array of 10 MySerializedObjects with ascending numbers
         MySerializedObject[] objs = new MySerializedObject[numObjs];
         for (int i = 0; i < numObjs; ++i) {
            objs[i] = new MySerializedObject(0xAA + i); // Starting at AA
         }
         // Write the objects to file, one by one.
         for (int i = 0; i < numObjs; ++i) {
            out.writeObject(objs[i]);
         }
         // Write the entire array in one go.
         out.writeObject(objs);
         out.flush();
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read raws bytes and print in Hex
      try (BufferedInputStream in =
              new BufferedInputStream(
                 new FileInputStream(filename))) {
         int inByte;
         while ((inByte = in.read()) != -1) {
            System.out.printf("%02X ", inByte);   // Print Hex codes
         }
         System.out.printf("%n%n");
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read objects
      try (ObjectInputStream in =
              new ObjectInputStream(
                 new BufferedInputStream(
                    new FileInputStream(filename)))) {
         // Read back the objects, cast back to its original type.
         MySerializedObject objIn;
         for (int i = 0; i < numObjs; ++i) {
            objIn = (MySerializedObject)in.readObject();
            System.out.println(objIn.getNumber());
         }
         MySerializedObject[] objArrayIn;
         objArrayIn = (MySerializedObject[])in.readObject();
         for (MySerializedObject o : objArrayIn) {
            System.out.println(o.getNumber());
         }
      } catch (ClassNotFoundException|IOException ex) {  // JDK 7
         ex.printStackTrace();
      }
   }
}

class MySerializedObject implements Serializable {
   private int number;

   public MySerializedObject(int number) {
      this.number = number;
   }

   public int getNumber() {
      return number;
```

```
   }
}
```

```
AC ED 00 05 73 72 00 12 4D 79 53 65 72 69 61 6C 69 7A 65 64 4F 62 6A 65
63 74 1F 7B 91 BD 02 1C DC 30 02 00 01 49 00 06 6E 75 6D 62 65 72 78 70
00 00 00 AA 73 71 00 7E 00 00 00 00 00 AB 73 71 00 7E 00 00
00 00 00 AC 73 71 00 7E 00 00 00 00 00 AD 73 71 00 7E 00 00
00 00 00 AE 75 72 00 15 5B 4C 4D 79 53 65 72 69 61 6C 69 7A 65 64 4F 62
6A 65 63 74 3B 13 95 A0 51 BC 86 75 38 02 00 00 78 70 00 00 00 05 71 00
7E 00 01 71 00 7E 00 02 71 00 7E 00 03 71 00 7E 00 04 71 00 7E 00 05
```

[Check out these bytes!]

Primitive types and array are, by default, serializable.

The `ObjectInputStream` and `ObjectOutputStream` implement `DataInput` and `DataOutput` interface respectively. You can used methods such as `readInt()`, `readDouble()`, `writeInt()`, `writeDouble()` for reading and writing primitive types.

t r a n s & s t a t i c

- `static` fields are not serialized, as it belongs to the class instead of the particular instance to be serialized.
- To prevent certain fields from being serialized, mark them using the keyword `transient`. This could cut down the amount of data traffic.
- The `writeObject()` method writes out the class of the object, the class signature, and values of non-`static` and non-`transient` fields.

6 . 2 j a v a . i o . S e r & E x t e r n a l i z a b l e i n t e r f a c e s

When you create a class that might be serialized, the class must implement `java.io.Serializable` interface. The `Serializable` interface doesn't declare any methods. Empty interfaces such as `Serializable` are known as *tagging interfaces*. They identify implementing classes as having certain properties, without requiring those classes to actually implement any methods.

Most of the core Java classes implement `Serializable`, such as all the wrapper classes, collection classes, and GUI classes. In fact, the only core Java classes that do not implement `Serializable` are ones that should not be serialized. Arrays of primitives or serializable objects are themselves serializable.

W a r n i n g M e s s a g e " T h e s e r i a l i z a t i o n c l a s s d o e s n o t (A d v a n c e d)

This warning message is triggered because your class (such as `java.swing.JFrame`) implements the `java.io.Serializable` interface. This interface enables the object to be written out to an output stream *serially* (via method `writeObject()`); and read back into the program (via method `readObject()`). The serialization runtime uses a number (called `serialVersionUID`) to ensure that the object read into the program (during deserialization) is compatible with the class definition, and not belonging to another version. It throws an `InvalidClassException` otherwise.

You have these options:

1. Simply ignore this warning message. If a `serializable` class does not explicitly declare a `serialVersionUID`, then the serialization runtime will calculate a default `serialVersionUID` value for that class based on various aspects of the class.
2. Add a `serialVersionUID` (Recommended), e.g.

   ```
   private static final long serialVersionUID = 1L;  // verion 1
   ```

3. Suppress this particular warning via annotation `@SuppressWarmomgs` (in package `java.lang`) (JDK 1.5):

   ```
   @SuppressWarnings("serial")
   public class MyFrame extends JFrame { ..... }
   ```

j a v a . i o . E x t I e n r t n e a r l f i a z c æ b l e

The `Serializable` has a sub-interface called `Externalizable`, which you could used if you want to customize the way a class is serialized. Since `Externalizable` extends `Serializable`, it is also a `Serializable` and you could invoke `readObject()` and `writeObject()`.

`Externalizable` declares two `abstract` methods:

```
void writeExternal(ObjectOutput out) throws IOException
void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
```

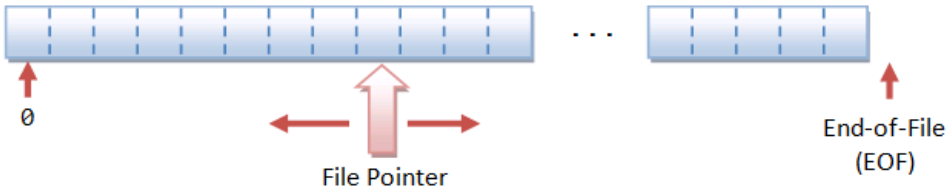
`ObjectOutput` and `ObjectInput` are interfaces that are implemented by `ObjectOutputStream` and `ObjectInputStream`, which define the `writeObject()` and `readObject()` methods, respectively. When an instance of `Externalizable` is passed to an `ObjectOutputStream`, the default serialization procedure is bypassed; instead, the stream calls the instance's `writeExternal()` method. Similarly, when an `ObjectInputStream` reads a `Exteranlizabled` instance, it uses `readExternal()` to reconstruct the instance.

`Externalizable` is useful if you want complete control on how your objects shall be serialized/deserialized. For example, you could encrypt sensitive data before the object is serialized.

7. Random Access Files

All the I/O streams covered so far are one-way streams. That is, they are either read-only input stream or write-only output stream. Furthermore, they are all sequential-access (or serial) streams, meant for reading and writing data sequentially. Nonetheless, it is sometimes necessary to read a file record directly as well as modifying existing records or inserting new records. The class `RandomAccessFile` provides supports for non-sequential, direct (or random) access to a disk file. `RandomAccessFile` is a two-way stream, supporting both input and output operations in the same stream.

`RandomAccessFile` can be treated as a huge byte array. You can use a file pointer (of type `long`), similar to array index, to access individual byte or group of bytes in primitive types (such as int and double). The file pointer is located at 0 when the file is opened. It advances automatically for every read and write operation by the number of bytes processed.



In constructing a `RandomAccessFile`, you can use flags `'r'` or `'rw'` to indicate whether the file is "read-only" or "read-write" access, e.g.,

```
RandomAccessFile f1 = new RandomAccessFile("filename", "r");
RandomAccessFile f2 = new RandomAccessFile("filename", "rw");
```

The following methods are available:

```
public void seek(long pos) throws IOException;
// Positions the file pointer for subsequent read/write operation.
public int skipBytes(int numBytes) throws IOException;
// Moves the file pointer forward by the specified number of bytes.
public long getFilePointer() throws IOException;
// Gets the position of the current file pointer, in bytes, from the beginning of the file.
public long length() throws IOException;
// Returns the length of this file.
```

`RandomAccessFile` does not inherit from `InputStream` or `OutputStream`. However, it implements `DataInput` and `DataOutput` interfaces (similar to `DataInputStream` and `DataOutputStream`). Therefore, you can use various methods to read/write primitive types to the file, e.g.,

```
public int readInt() throws IOException;
public double readDouble() throws IOException;
public void writeInt(int i) throws IOException;
public void writeDouble(double d) throws IOException;
```

Example Read and write records from a `RandomAccessFile`. (A student file consists of student record of name (`String`) and id (`int`)).

[PENDING]

8. Compressed I/O Streams

The classes `ZipInputStream` and `ZipOutputStream` (in package `java.util`) support reading and writing of compressed data in ZIP format. The classes `GZIPInputStream` and `GZIPOutputStream` (in package `java.util`) support reading and writing of compressed data in GZIP format.

Example Reading and writing ZIP files

[@PENDING]

Example Reading and writing JAR files

[@PENDING]

9. Formatted Text I/O - java.util.Formatter & java.util.Scanner

9.1 Formatted Text Input with java.util.Scanner

JDK 1.5 introduces `java.util.Scanner` class, which greatly simplifies formatted text input from input source (e.g., files, keyboard, network). `Scanner`, as the name implied, is a simple text scanner which can parse the input text into primitive types and strings using regular expressions. It first breaks the text input into *tokens* using a delimiter pattern, which is by default the white spaces (blank, tab and newline). The tokens may then be converted into primitive values of different types using the various `nextXxx()` methods (`nextInt()`, `nextByte()`, `nextShort()`, `nextLong()`, `nextFloat()`, `nextDouble()`, `nextBoolean()`, `next()` for `String`, and `nextLine()` for an input line). You can also use the `hasNextXxx()` methods to check for the availability of a desired input.

The commonly-used constructors are as follows. You can construct a `Scanner` to parse a byte-based `InputStream` (e.g., `System.in`), a disk file, or

a given `String`.

```
// Scanner piped from a disk File
public Scanner(File source) throws FileNotFoundException
public Scanner(File source, String charsetName) throws FileNotFoundException
// Scanner piped from a byte-based InputStream, e.g., System.in
public Scanner(InputStream source)
public Scanner(InputStream source, String charsetName)
// Scanner piped from the given source string (NOT filename string)
public Scanner(String source)
```

For examples,

```
// Construct a Scanner to parse an int from keyboard
Scanner in1 = new Scanner(System.in);
int i = in1.nextInt();

// Construct a Scanner to parse all doubles from a disk file
Scanner in2 = new Scanner(new File("in.txt"));  // need to handle FileNotFoundException
while (in2.hasNextDouble()) {
   double d = in.nextDouble();
}

// Construct a Scanner to parse a given text string
Scanner in3 = new Scanner("This is the input text String");
while (in3.hasNext()) {
   String s = in.next();
}
```

Example 1 The most common usage of `Scanner` is to read primitive types and `String` form the keyboard (`System.in`), as follows:

```
import java.util.Scanner;
public class TestScannerSystemIn {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);

      System.out.print("Enter an integer: ");
      int anInt = in.nextInt();
      System.out.println("You entered " + anInt);

      System.out.print("Enter a floating-point number: ");
      double aDouble = in.nextDouble();
      System.out.println("You entered " + aDouble);

      System.out.print("Enter 2 words: ");
      String word1 = in.next();  // read a string delimited by white space
      String word2 = in.next();  // read a string delimited by white space
      System.out.println("You entered " + word1 + " " + word2);

      in.nextLine();   // flush the "enter" before the next readLine()

      System.out.print("Enter a line: ");
      String line = in.nextLine();  // read a string up to line delimiter
      System.out.println("You entered " + line);
   }
}
```

The `nextXxx()` methods throw `InputMismatchException` if the next token does not match the type to be parsed.

Example 2 You can easily modify the above program to read the inputs from a text file, instead of keyboard (`System.in`).

```
import java.util.Scanner;
import java.io.*;
public class TestScannerFile {
   public static void main(String[] args) throws FileNotFoundException {
      Scanner in = new Scanner(new File("in.txt"));

      System.out.print("Enter an integer: ");
      int anInt = in.nextInt();
      System.out.println("You entered " + anInt);

      System.out.print("Enter a floating-point number: ");
      double aDouble = in.nextDouble();
      System.out.println("You entered " + aDouble);

      System.out.print("Enter 2 words: ");
      String word1 = in.next();  // read a string delimited by white space
      String word2 = in.next();  // read a string delimited by white space
      System.out.println("You entered " + word1 + " " + word2);
```

```
      in.nextLine();   // flush the "enter" before the next readLine()

      System.out.print("Enter a line: ");
      String line = in.nextLine();  // read a string up to line delimiter
      System.out.println("You entered " + line);
   }
}
```

Prepare the input file "`in.txt`" as follows:

```
123
44.55
first second
this is a test
```

`nextXxx()` and `nextXxx()`

The `Scanner` class implements `iterator<String>` interface. You can use `hasNext()` coupled with `next()` to iterate through all the `String` tokens. You can also directly iterate through the primitive types via methods `hasNextXxx()` and `nextXxx()`. `Xxx` includes all primitive types (`byte`, `short`, `int`, `long`, `float`, `double` and `boolean`), `BigInteger`, and `BigNumber`. `char` is not included but can be retrieved from `String` via `charAt()`.

Delimiter

Instead of the default white spaces as the delimiter, you can set the delimiter to a chosen regular expression via these methods:

```
public Pattern delimiter()    // Returns the current delimiter Regexe Pattern
public Scanner useDelimiter(Pattern pattern)   // Sets the delimiter Regexe Pattern
public Scanner useDelimiter(String pattern)
```

Example 3: Customized Token delimiter

```
import java.util.Scanner;
public class ScannerWithDelimiter {
   public static void main(String[] args) {
      Scanner in = new Scanner("one apple 2 apple red apple big apple 5.5 apple");
      // Zero or more whitespace, followed by 'apple', followed by zero or more whitespace.
      in.useDelimiter("\\s*apple\\s*");
      // The delimiter breaks the input into tokens {"one", "2", "red", "big", "5.5"}.
      System.out.println(in.next());
      System.out.println(in.nextInt());      // parses text into int
      System.out.println(in.next());
      System.out.println(in.next());
      System.out.println(in.nextDouble());   // parses text into double
   }
}
```

The regular expression `\s*apple\s*` matches zero or more white spaces (`\s*`) followed by "apple" followed by zero or more white spaces (`\s*`). An additional backslash (\) is needed to use a backslash (\) in Java `String`'s literal. Read "Regular Expression" for more details.

Regexe Pattern Matching

You can use the following methods to find the next occurrence of the specified pattern using regular expressions:

```
// Find the next occurrence of a pattern, ignoring delimiters
public String findInLine(Pattern pattern)
public String findInLine(String pattern)
public String findWithinHorizon(Pattern pattern, int horizon)
public String findWithinHorizon(String pattern, int horizon)
// Skips input that matches the specified pattern, ignoring delimiters
public Scanner skip(Pattern pattern)
public Scanner skip(String pattern)
```

Charset

By default, `Scanner` uses the default charset to read the character from the input source. You can ask `Scanner` to read text file which is encoded using a particular charset, by providing the charset name.

Example 4:

```
import java.util.*;
import java.io.*;
public class TestScannerTextFile {
   public static void main(String[] args) throws FileNotFoundException {
      String filename = "test_scanner.txt";
      String message = "Hi,您好!\n";   // with non-ASCII chars
      // Create a Text file in UTF-8
      // Can also use formatter (see below)
```

```
      try (BufferedWriter out =
              new BufferedWriter(
                 new OutputStreamWriter(
                    new FileOutputStream(filename), "UTF-8"))) {
         out.write("12345 55.66\n");
         out.write(message);
         out.flush();
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read raws bytes and print in Hex
      try (BufferedInputStream in =
              new BufferedInputStream(
                 new FileInputStream(filename))) {
         int inByte;
         while ((inByte = in.read()) != -1) {
            System.out.printf("%02X ", inByte);   // Print Hex codes
         }
         System.out.println();
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Open a text file, specifying the charset for the file encoding
      Scanner in = new Scanner(new File(filename), "UTF-8");
      System.out.println(in.nextInt());
      System.out.println(in.nextFloat());
      in.nextLine();  // Flush a newline
      System.out.println(in.nextLine());
   }
}
```

```
31 32 33 34 35 20 35 35 2E 36 36 0A 48 69 2C E6 82 A8 E5 A5 BD 21 0A
1  2  3  4  5  SP 5  5  .  6  6  LF H  i  ,  您       好       !  LF [UTF-8]

12345
55.66
Hi,您好!
```

9.2 Formatted-Type Printing with printf()

JDK 1.5 introduces C-like `printf()` method (in classes `java.io.PrintSteam` and `java.io.PrintWriter`) for formatted-text printing.

To write formatted-text to console (`System.out` or `System.err`), you could simply use the convenience methods `System.out.printf()` or `System.out.format()`. `printf()` takes this syntax:

```
public PrintStream|PrintWriter printf(String formatSpecifier, Object... args)
public PrintStream|PrintWriter printf(Locale locale, String formatSpecifier, Object... args)
```

`printf()` takes a variable number (zero or more) of arguments (or varargs). Varargs was introduced in JDK 1.5 (that is the reason Java cannot support `printf()` earlier).

`printf()` can be called from `System.out` or `System.err` (which are `PrintStreams`). For example,

```
System.out.printf("Hello %4d %6.2f %s, and%n   Hello again%n", 123, 5.5, "Hello");
```

```
Hello  123   5.50 Hello, and
   Hello again
```

You can also use the `System.out.format()` method, which is identitical to `System.out.printf()`.

Format Specifiers

A format specifier begins with `'%'` and ends with a conversion-type character (e.g. `"%d"` for integer, `"%f"` for `float` and `double`), with optional parameters in between, as follows:

```
%[argument_position$][flag(s)][width][.precision]conversion-type-character
```

- The optional *argument_position* specifies the position of the argument in the argument list. The first argument is `"1$"`, second argument is `"2$"`, and so on.
- The optional *width* indicates the minimum number of characters to be output.
- The optional *precision* restricts the number of characters (or number of decimal places for float-point numbers).
- The mandatory *conversion-type-character* indicates how the argument should be formatted. For examples: `'b'`, `'B'` (`boolean`), `'h'`, `'H'` (hex string), `'s'`, `'S'` (string), `'c'`, `'C'` (character), `'d'` (decimal integer), `'o'` (octal integer), `'x'`, `'X'` (hexadecimal integer), `'e'`, `'E'` (float-point

number in scientific notation), `'f'` (floating-point number), `'%'` (percent sign). The uppercase conversion code (e.g., `'S'`) formats the texts in uppercase.

- Flag: `'-'` (left-justified), `'+'` (include sign), `' '` (include leading space), `'0'` (zero-padded), `','` (include grouping separator). `'('` (negative value in parentheses), `'#'` (alternative form).

Read JDK API `java.util.Formatter`'s "Format String Syntax" for details on format specifiers.

Examples

```
System.out.printf("%2$2d %3$2d %1$2d%n", 1, 12, 123, 1234);
```

```
12 123  1
```

```
System.out.printf(Locale.FRANCE, "e = %+10.4f%n", Math.PI);
```

```
e =    +3,1416
```

```
System.out.printf("Revenue: $ %(,.2f, Profit: $ %(,.2f%n", 12345.6, -1234.5678);
```

```
Revenue: $ 12,345.60, Profit: $ (1,234.57)
```

9.3 Formatted Text Output with `java.util.Formatter`

JDK 1.5 introduced `java.util.Scanner` for formatted text input. It also introduced `java.util.Formatter` for formatted text output.

A `Formatter` is an interpreter for `printf`-style format strings. It supports layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output, via the format specifiers.

The `Formatter` has the following constructors:

```
public Formatter(Appendable a);    // StringBuffer and StringBuilder implement Appendable
public Formatter(Appendable a, Locale l);
public Formatter(String filename)
public Formatter(String fileName, String charsetName)
public Formatter(File file)
public Formatter(File file, String charsetName)
public Formatter(OutputStream os)
public Formatter(PrintStream ps)
```

`Formatter` supports `StringBuffer`/`StringBuilder` (as `Appendable`) as output sink. It does not support `String`, probably because `String` is immutable.

The `format()` method can be used to write formatted text output:

```
public Formatter format(String formatSpecifier, Object... args)
public Formatter format(Locale locale, String formatSpecifier, Object... args)
```

Notice that the method `format()` has the same syntax as the method `printf()`, using the same set of format specifier as `printf()`.

Other methods are:

```
public void flush()   // flush out all the buffered output
public void close()
```

Example: Using a `StringBuilder` (which implements `Appendable`) as the output sink for the `Formatter`.

```
1  import java.util.Formatter;
2  import java.util.Locale;
3
4  public class TestFormatter {
5     public static void main(String[] args) {
6
7     // Use a StringBuilder (Appandable) as the output sink for the Formatter
8     StringBuilder sb = new StringBuilder();
9     Formatter formatter = new Formatter(sb, Locale.US);
10
11    // Re-order output.
12    formatter.format("%4$2s %3$2s %2$2s %1$2s", "a", "b", "c", "d");
13    System.out.println(sb);    // -> " d  c  b  a"
14
15    // Use the optional locale as the first argument.
16    sb.delete(0, sb.length());
17    formatter.format(Locale.FRANCE, "e = %+10.4f", Math.E);
18    System.out.println(sb);    // -> "e =    +2,7183"
19
20    // Try negative number with '(' flag and group separator.
21    sb.delete(0, sb.length());
```

```
   formatter.format("Net gained or lost: $ %(,.2f", -1234.567);
   System.out.println(sb);    // -> "Net gained or lost: $ (1,234.57)"
   }
}
```

Read JDK API `java.util.Formatter`'s "Format String Syntax" for details on format specifiers.

Example 2: Setting the charset for `Formatter`'s output.

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

public class FormatterTest {
   public static void main(String[] args) {
      String filename = "formatter-out.txt";
      String message = "Hi,您好!";

      // Create a text file in "UTF-8"
      try (Formatter out = new Formatter(filename, "UTF-8")) {
         out.format("%4d %6.2f %s%n", 0xAA, 55.66, message);
         out.flush();
      } catch (UnsupportedEncodingException|FileNotFoundException ex) {
         ex.printStackTrace();
      }

      // Read raws bytes and print in Hex
      try (BufferedInputStream in =
              new BufferedInputStream(
                 new FileInputStream(filename))) {
         int inByte;
         while ((inByte = in.read()) != -1) {
            System.out.printf("%02X ", inByte);   // Print Hex codes
         }
         System.out.println();
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Set up text file input using a scanner and read records
      try (Scanner in = new Scanner(new File(filename), "UTF-8")) {
         System.out.println(in.nextInt());
         System.out.println(in.nextDouble());
         System.out.println(in.next());
      } catch (FileNotFoundException ex) {
         ex.printStackTrace();
      }
   }
}
```

```
20 31 37 30 20 20 35 35 2E 36 36 20 48 69 2C E6 82 A8 E5 A5 BD 21 0A
170         SP 55.66            SP H  i  ,  您       好       !  LF

170
55.66
Hi,您好!
```

9.4 String.format()

The `Formatter` with `StringBuilder` as the output sink allows you to build up a formatted string progressively. To produce a simple formatted `String`, you can simply use the `static` method `String.format()`. This is handy in the `toString()` method, which is required to return a `String`. For example,

```
public class Time {
   private int hour, minute, second;

   public Time(int hour, int minute, int second) {
      this.hour = hour;
      this.minute = minute;
      this.second = second;
   }

   // Returns a description in "HH:MM:SS"
   public String toString() {
      return String.format("%02d:%02d:%02d", hour, minute, second);
   }

   // Test main()
   public static void main(String[] args) {
```

```
17          Time t = new Time(1, 2, 3);
18          System.out.println(t);   // 01:02:03
19       }
20    }
```

10. File I/O in JDK 1.7

[This section was extracted from the Java Online Tutorial and JDK 7 API.]

JDK 1.7 greatly enhances supports for file I/O via new package `java.nio.file` and its associated packages.

10.1 Introduction to `java.nio.file.Path`

A *path string* could be used to locate a *file*, a *directory* or a *symbolic link*. A *symbolic link* (or *symlink*) is a special file that references another file. A path string is system dependent, e.g., "`c:\myproject\java\Hello.java`" in Windows or "`/myproject/java/Hello.java`" in Unix. Windows uses back-slash '`\`' as the directory separator; while Unixes use forward-slash '`/`'. Windows uses semi-colon '`;`' as path separator; while Unixes use colon '`:`'. The "`c:\`" or "`\`" is called the *root*. Windows supports multiple roots, each maps to a drive (e.g., "`c:\`", "`d:\`"). Unix has single root ("`\`"). A path could be *absolute* (beginning from the root) or *relative* (which is relative to the current working directory). Special notations "`.`" and "`..`" denote the current directory and the parent directory, respectively.

A `java.nio.file.Path` instance specifies the location of a file, or a directory, or a symbolic link. `Path` replaces `java.io.File` (of the standard I/O), which is less versatile and buggy.

10.2 Helper Class `java.nio.file.Paths`

To create a `Path`, use the `static` method `get()` of the helper class `java.nio.file.Paths`. The helper class `Paths` contains exclusively `static` methods for creating `Path` objects. `Paths.get()` returns a `Path` object by converting a given path string or `URI`.

```
public static Path get(String first, String... more)
// This method accepts variable number of arguments (varargs).
// It converts a path string, or a sequence of strings that when joined form a path string, to a Path object.
// The location of the Path may or may not exist.

public static Path get(URI uri)
// Converts the given URI to a Path object.
```

For example,

```
Path p1 = Paths.get("in.txt");      // A file in current directory, relative
Path p2 = Paths.get("c:\\myproejct\\java\\Hello.java");  // File, absolute, need escape sequence for '\'
Path p3 = Paths.get("/use/local");  // A directory
```

As the directory-separator is system dependent, for writing portable and more flexible program, it is recommended to use an existing `Path` instance as an anchor to resolve the filename, e.g.,

```
Path dir = ...
Path file = dir.resolve("filename");  // Joins the filename to the dirname with a directory-separator
```

A `Path` can be broken down as root, level-0, level-1, ... The method `getNameCount()` returns the levels. The method `getName(i)` returns the name of level-i. For example,

```
import java.nio.file.*;
public class PathInfo {
   public static void main(String[] args) {
      // Windows
      Path path = Paths.get("D:\\myproject\\java\\test\\Hello.java");
      // Unix/Mac
      //Path path = Paths.get("/myproject/java/test/Hello.java");

      // Print Path Info
      System.out.println("toString:    " + path.toString());    // D:\myproject\java\test\Hello.java
      System.out.println("getFileName: " + path.getFileName()); // Hello.java
      System.out.println("getParent: " + path.getParent());     // D:\myproject\java\test
      System.out.println("getRoot: " + path.getRoot());         // D:\

      // root, level-0, level-1, ...
      int nameCount = path.getNameCount();
      System.out.println("getNameCount: " + nameCount);   // 4
      for (int i = 0; i < nameCount; ++i) {
         System.out.println("getName(" + i + "): " + path.getName(i)); // (0)myproject, (1)java,
      }                                                                 // (2) test, (3) Hello.java
      System.out.println("subpath(0,2): " + path.subpath(0,2));  // myproject\java
      System.out.println("subpath(1,4): " + path.subpath(1,4));  // java\test\Hello.java
   }
}
```

10.3 Hjælpeklasse java.nio.file.Files

The class `Files` contains exclusively `static` methods for file, directory and symlink operations such as create, delete, read, write, copy, move, etc.

Properties of a File/Directory

You can use `static boolean` methods `Files.exists(Path)` and `File.notExists(Path)` to verify if a given `Path` exists or does not exist (as a file, directory or symlink). A `Path` could be verified to exist, or not exist, or unknown (e.g., the program does not have access to the file). If the status is unknown, the `exists()` and `noExists()` returns `false`.

You could also use `static boolean` methods `Files.isDirectory(Path)`, `Files.isRegularFile(Path)` and `Files.isSymbolicLink(Path)` to verify whether a `Path` locates a file, directory, or symlink.

Many of these methods take an optional second argument of `LinkOption`, which is applicable for symlink only. For example, `LinkOption.NOFOLLOW_LINKS` specifies do not follow the symlink.

```
public static long size(Path path)  // Returns the size of the file

public static boolean exists(Path path, LinkOption... options)
// Verifies whether the given Path exists, as a file/directory/symlink.
// It returns false if the file does not exist or status is unknown.
// LinkOption specifies how symlink should be handled,
//   e.g., NOFOLLOW_LINKS: Do not follow symlinks.
public static boolean notExists(Path path, LinkOption... options)     // Not exists?

public static boolean isDirectory(Path path, LinkOption... options)   // a directory?
public static boolean isRegularFile(Path path, LinkOption... options) // a file?
public static boolean isSymbolicLink(Path path)                       // a symblink?

public static boolean isReadable(Path path)    // readable?
public static boolean isWritable(Path path)    // writable?
public static boolean isExecutable(Path path)  // executable?
```

For example, to get the size of a file in JDK 1.7:

```
// JDK 7: Use static method java.nio.file.Files.size(path)
Path path = Paths.get("test.jpg");
System.out.println("Files.Size(): " + Files.size(path));

// Pre-JDK 7: Use java.io.File method aFile.length()
File file = new File("test.jpg");
System.out.println("File.length():" + file.length());
```

Deleting a File/Directory

You can use `static` methods `delete(Path)` to delete a file or directory. Directory can be deleted only if it is empty. A `boolean` method `deleteIfExists()` is also available.

```
public static void delete(Path path) throws IOException
public static boolean deleteIfExists(Path path) throws IOException
```

Copying/Moving a File/Directory

You can use `static` methods `copy(Path, Path, CopyOption)` or `move(Path, Path, CopyOption)` to copy or move a file or directory. The methods return the target `Path`.

The methods accepts an *optional* third argument of `CopyOption`. For examples: `CopyOption.REPLACE_EXISTING` replaces the target if it exists; `CopyOption.COPY_ATTRIBUTES` copies the file attributes such as the dates; `CopyOption.NOFOLLOW_LINKS` specifies not to follow symlinks.

```
public static Path copy(Path source, Path target, CopyOption... options) throws IOException
public static Path move(Path source, Path target, CopyOption... options) throws IOException
```

To copy a file into another directory:

```
Path source = ...
Path targetDir = ...
Files.copy(source, targetDir.resolve(source.getFileName());
```

Reading/Writing Small Files

For small files, you can use `static` methods `Files.readAllBytes(Path)` (byte-based) and `Files.readAllLines(Path, Charset)` (character-based) to read the entire file. You can use `Files.write(Path, byte[])` (byte-based) or `Files.write(Path, Iterable, Charset)` (character-based) to write to a file.

The optional `OpenOption` includes: `WRITE`, `APPEND`, `TRUNCATE_EXISTING` (truncates the file to zero bytes), `CREATE_NEW` (creates a new file and

throws an exception if the file already exists), CREATE (opens the file if it exists or creates a new file if it does not), among others.

Byte-based Operation

```
public static byte[] readAllBytes(Path path) throws IOException
// Reads all bytes and returns a byte array.
public static Path write(Path path, byte[] bytes, OpenOption... options) throws IOException
// Default options are: CREATE, TRUNCATE_EXISTING, and WRITE
```

Character-based Operation

```
public static List<String> readAllLines(Path path, Charset cs) throws IOException
// Reads all lines and returns a list of String.
// Line terminator could be "\n", "\r\n" or "\r".
public static Path write(Path path, Iterable<? extends CharSequence> lines,
                         Charset cs, OpenOption... options) throws IOException
```

Example The following example write a small text file (in "UTF-8"), and read the entire file back as bytes as well as characters.

```
import java.io.*;
import java.nio.file.*;
import java.nio.charset.*;
import java.util.*;

public class SmallFileIOJDK7 {
   public static void main(String[] args) {
      String fileStr = "small_file.txt";
      Path path = Paths.get(fileStr);

      // Strings with unicode characters
      List<String> lines = new ArrayList<String>();
      lines.add("Hi,您好!");
      lines.add("Hello,吃饱了没有?");

      // Write to text file in UTF-8
      try {
         Files.write(path, lines, Charset.forName("UTF-8"));
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read the whole file as bytes
      byte[] bytes;
      try {
         bytes = Files.readAllBytes(path);
         for (byte aByte: bytes) {
            System.out.printf("%02X ", aByte);
         }
         System.out.printf("%n%n");
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read the whole file as characters
      List<String> inLines;
      try {
         inLines = Files.readAllLines(path, Charset.forName("UTF-8"));
         for (String aLine: inLines) {
            for (int i = 0; i < aLine.length(); ++i) {
               char charOut = aLine.charAt(i);
               System.out.printf("[%d]'%c'(%04X) ", (i+1), charOut, (int)charOut);
            }
            System.out.println();
         }
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

```
48 69 2C E6 82 A8 E5 A5 BD 21 0D 0A
H  i  ,  您       好       !  CR LF
48 65 6C 6C 6F 2C E5 90 83 E9 A5 B1 E4 BA 86 E6 B2 A1 E6 9C 89 3F 0D 0A
H  e  l  l  o  ,  吃       饱       了       没       有       ?  CR LF

[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)

[1]'H'(0048) [2]'e'(0065) [3]'l'(006C) [4]'l'(006C) [5]'o'(006F) [6]','(002C)
[7]'吃'(5403) [8]'饱'(9971) [9]'了'(4E86) [10]'没'(6CA1) [11]'有'(6709) [12]'?'(003F)
```

Buffered Character-based I/O for Text Files

For Reading, use `Files.newBufferedReader(Path, Charset)` method to open a text file, which returns a `BufferedReader`. Use `BufferedReader.readLine()` to read a line, `read()` to read a char, or `read(char[] cbuf, int off, int len)` to read into a char-array.

For Writing, use the `Files.newBufferedWriter(Path, Charset, OpenOption...)` method to open a output text file, which returns a `BufferedWriter`. Use `BufferedWriter.write(int c)` to write a character, `write(char[] cbuf, int off, int len)` or `write(String s, int off, int len)` to write characters.

```
public static BufferedReader newBufferedReader(Path path, Charset cs) throws IOException

public static BufferedWriter newBufferedWriter(Path path, Charset cs, OpenOption... options)
    throws IOException
```

Byte-Based Stream I/O

Use `Files.newInputStream(Path, OpenOption...)` to allocate an `InputStream` for reading raw bytes; and `Files.newOutputStream(Path, OpenOption...)` to allocate an `OutputStream` for writing. The `InputStream` and `OutputStream` returned are not buffered.

Example: Similar to the previous program which read/write the entire file, this program read/write via Buffered I/O.

```
import java.io.*;
import java.nio.file.*;
import java.nio.charset.*;
import java.util.*;

public class BufferedFileIOJDK7 {
   public static void main(String[] args) {
      String fileStr = "buffered_file.txt";
      Path path = Paths.get(fileStr);

      // Strings with unicode characters
      List<String> lines = new ArrayList<String>();
      lines.add("Hi,您好!");
      lines.add("Hello,吃饱了没有?");
      Charset charset = Charset.forName("UTF-8");

      // Write to text file (encoded in UTF-8)
      try (BufferedWriter out = Files.newBufferedWriter(path, charset)) {
         for (String line: lines) {
            out.write(line, 0, line.length());
         }
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read the file as bytes
      try (BufferedInputStream in = new BufferedInputStream(Files.newInputStream(path))) {
         int inByte;
         while ((inByte = in.read()) != -1) {
            System.out.printf("%02X ", inByte);
         }
         System.out.printf("%n%n");
      } catch (IOException ex) {
         ex.printStackTrace();
      }

      // Read the file as characters
      String inLine;
      try (BufferedReader in = Files.newBufferedReader(path, charset)) {
         while ((inLine = in.readLine()) != null) {
            for (int i = 0; i < inLine.length(); ++i) {
               char aChar = inLine.charAt(i);
               System.out.printf("[%d]'%c'(%04X) ", (i+1), aChar, (int)aChar);
            }
            System.out.println();
         }
      } catch (IOException ex) {
         ex.printStackTrace();
      }
   }
}
```

```
48 69 2C E6 82 A8 E5 A5 BD 21
H  i  ,  您       好       !
48 65 6C 6C 6F 2C E5 90 83 E9 A5 B1 E4 BA 86 E6 B2 A1 E6 9C 89 3F
```

```
H e l l o , 吃 饱 了 没 有 ?

[1]'H'(0048) [2]'i'(0069) [3]','(002C) [4]'您'(60A8) [5]'好'(597D) [6]'!'(0021)
[7]'H'(0048) [8]'e'(0065) [9]'l'(006C) [10]'l'(006C) [11]'o'(006F) [12]','(002C)
[13]'吃'(5403) [14]'饱'(9971) [15]'了'(4E86) [16]'没'(6CA1) [17]'有'(6709) [18]'?'(003F)
```

Creating a New File / Directory / Symlink

Beside using the `Files.write()` method with `OpenOption` of `CREATE` or `CREATE_NEW`, you can also use `Files.createFile()` method to create an empty file. You can use the default file attributes or optionally define the initial attributes of the file.

```
public static Path createFile(Path path, FileAttribute<?>... attrs)
```

The `FileAttribute` includes: [TODO]

- DOS:
- Unixes: Nine file permissions: read, write, and execute permissions for the file owner, members of the same group, and "everyone else", e.g., "`rwxr-x---`".

Example: [TODO]

Similary, you can create a new directory, symlink as follows:

```
public static Path createDirectory(Path dir, FileAttribute<?>... attrs) throws IOException
// Creates a new directory.

public static Path createDirectories(Path dir, FileAttribute<?>... attrs) throws IOException
// Creates a directory by creating all nonexistent "parent" directories first

public public static Path createSymbolicLink(Path link, Path target,
   FileAttribute<?>... attrs) throws IOException
// Creates a symbolic link to a target
```

Example:

```
try {
   Files.createDirectory(Paths.get("d:\\temp\\test"));
      // With default attribute
      // FileAlreadyExistsException if already exists
   Files.createDirectory(Paths.get("d:\\temp\\test1\\test2"));
      // NoSuchFileException if parent does not exist
   Files.createDirectories(Paths.get("d:\\temp\\test1\\test2"));
      // Also create the parent directory
} catch (IOException ex) {
   ex.printStackTrace();
}
```

Creating a Temporary File / Directory

```
public static Path createTempFile(Path dir, String prefix, String suffix,
      FileAttribute<?>... attrs) throws IOException
// Creates a new empty file in the specified dir,
//  using the given prefix and suffix to generate its name.

public static Path createTempFile(String prefix, String suffix,
      FileAttribute<?>... attrs) throws IOException
// Creates an empty file in the default temporary-file directory,
//  using the given prefix and suffix to generate its name.

public static Path createTempDirectory(
      Path dir, String prefix, FileAttribute<?>... attrs) throws IOException

public static Path createTempDirectory(
      String prefix, FileAttribute<?>... attrs) throws IOException
```

Example: [TODO]

10.4 File Attributes

You can use one of the `Files.readAttributes()` methods to read all the basic attribute of a path.

```
public static Map<String,Object> readAttributes(
      Path path, String attributes, LinkOption... options) throws IOException
// Read Reads a set of file attributes.

public static Object getAttribute(
      Path path, String attribute, LinkOption... options) throws IOException
// Get a given attribute
```

```
public static Path setAttribute(
      Path path, String attribute, Object value, LinkOption... options) throws IOException
// Set a given attribute
```

10.5 FileChannel

```
public static SeekableByteChannel newByteChannel(
      Path path, OpenOption... options) throws IOException
// Opens or creates a file, returning a seekable byte channel to access the file.

public static SeekableByteChannel newByteChannel(
      Path path, Set<? extends OpenOption> options, FileAttribute<?>... attrs)
      throws IOException
// Opens or creates a file, returning a seekable byte channel to access the file.
```

10.6 Random Access File

The Interface `SeekableByteChannel` supports random access.

```
pubic long position()
// Returns this channel's position.
public SeekableByteChannel position(long newPosition)
// Sets this channel's position.

public int read(ByteBuffer dest)
// Reads a sequence of bytes from this channel into the given ByteBuffer.
public int write(ByteBuffer source)
// Writes a sequence of bytes to this channel from the given ByteBuffer.

public long size()
// Returns the current size of entity to which this channel is connected.
public SeekableByteChannel truncate(long size)
// Truncates the entity, to which this channel is connected, to the given size.
```

10.7 Directory Operations

List all root directories

```
// Print the root directories for the default file system
Iterable<Path> rootDirs = FileSystems.getDefault().getRootDirectories();
for (Path rootDir : rootDirs) {
   System.out.println(rootDir);
}
```

Listing a directory

You can list the contents of a directory by using the `Files.newDirectoryStream(Path)` method. The returned `DirectoryStream` object implements `Iterable`. You can interate thru the entries with for-each loop.

```
public static DirectoryStream<Path> newDirectoryStream(Path dir) throws IOException
```

Example List the contents of a directory

```
// List the contents of a directory
Path dir = Paths.get(".");
try (DirectoryStream<Path> dirStream = Files.newDirectoryStream(dir)) {
   for (Path entry : dirStream) {
      System.out.println(entry.getFileName());  // Filename only
      System.out.println(entry.toString());     // Full-path name
   }
} catch (IOException | DirectoryIteratorException ex) {
   ex.printStackTrace();
}
```

In addition, you can include a glob pattern to filter the entries.

```
public static DirectoryStream<Path> newDirectoryStream(Path dir, String glob)
      throws IOException
```

Example List the contents of a directory filtered by a glob.

```
// List the contents of a directory filtered by a glob pattern
Path dir = Paths.get(".");
try (DirectoryStream<Path> dirStream
        = Files.newDirectoryStream(dir, "h*.{java,class,txt}")) {
```

```
        // begins with 'h' and ends with .java or .class or .txt
   for (Path entry : dirStream) {
      System.out.println(entry.getFileName());  // Filename only
      System.out.println(entry.toString());     // Full-path name
   }
} catch (IOException ex) {
   ex.printStackTrace();
}
```

Glob

A *glob* is a subset of regular expression.

- `'*'` matches zero or more character (0+).
- `'?'` matches any one character.
- `'**'` behaves like `'*'`, but can cross directory boundary, for matching on full path.
- `{...,...,...}` encloses a set of sub-patterns separated by ','.
- `[...]` encloses a set of single character or a range of character with `'-'`, e.g., [aeiou], [a-z], [0-9].
- Any other character matches itself. To match `'*'`, `'?'` or special characters, prefix with `'\'`.

For example: "`h*.{java,class,txt}`" matches entry starts with "h" and ends with "`.java`", "`.class`", or "`.txt`".

You can also write your own codes to filter the entries.

```
public static DirectoryStream<Path> newDirectoryStream(
      Path dir, DirectoryStream.Filter<? super Path> filter) throws IOException
```

The interface `DirectoryStream.Filter<T>` declares one `abstract boolean` method `accept()`, which will be call-back for each entry. Those entries that resulted in false `accept()` will be discarded.

```
public boolean accept(T entry) throws IOException
```

Example The following program uses an anonymous instance of an anonymous `DirectoryStream.Filter` sub-class to filter the `DirectoryStream`. The call-back method `accept()` returns `true` for regular files, and discards the rest. Take note that this filtering criterion cannot be implemented in a glob-pattern.

```
// List the contents of a directory with a custom filter
Path dir = Paths.get("d:\\temp");
try (DirectoryStream<Path> dirStream = Files.newDirectoryStream(dir,
         new DirectoryStream.Filter<Path>() {   // anonymous inner class
            public boolean accept(Path entry) throws IOException {
                  return (Files.isRegularFile(entry));  // regular file only
            }
         })) {
   for (Path entry : dirStream) {
      System.out.println(entry.getFileName());  // Filename only
      System.out.println(entry.toString());     // Full-path name
   }
} catch (IOException ex) {
   ex.printStackTrace();
}
```

10.8 Walking Files with Files.walkFileTree()

You can use `static` method `Files.walkFileTree()` to recursively walk thru all the files from a starting directory.

First of all, you need to create an object that implements interface `FileVisitor<? super Path>`, which declares these `abstract` methods:

```
public FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs) throws IOException
// Invoked for a directory before entries in the directory are visited.
// If this method returns CONTINUE, then entries in the directory are visited.
// If this method returns SKIP_SUBTREE or SKIP_SIBLINGS then entries in the directory
//   (and any descendants) will not be visited.

public FileVisitResult postVisitDirectory(T dir, IOException ex) throws IOException
// Invoked for a directory after entries in the directory, and all of their descendants,
//   have been visited. This method is also invoked when iteration of the directory
//   completes prematurely (by a visitFile method returning SKIP_SIBLINGS,
//   or an I/O error when iterating over the directory).

public FileVisitResult visitFile(T file, BasicFileAttributes attrs) throws IOException
// Invoked for a file/symlink in a directory.

public FileVisitResult visitFileFailed(T file, IOException ex) throws IOException
// Invoked for a file that could not be visited. This method is invoked
//   if the file's attributes could not be read, the file is a directory
```

```
//   that could not be opened, and other reasons.
```

These methods return an `enum` of `FileVisitResult`, which could take values of `CONTINUE`, `TERMINATE`, `SKIP_SUBTREES`, `SKIP_SIBLINGS`.

Instead of implementing `FileVisitor` interface, you could also extend from superclass `SimpleFileVisitor`, and override the selected methods.

There are two versions of `walkFileTree()`. The first version take a starting directory and a `FileVisitor`, and transerve through all the levels, without following the symlinks.

```
public static Path walkFileTree(Path startDir, FileVisitor<? super Path> visitor) throws IOException
```

E x a m p l e :

```
import java.util.EnumSet;
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import static java.nio.file.FileVisitResult.*;

public class WalkFileTreeTest extends SimpleFileVisitor<Path> {

   // Print the directory visited.
   @Override
   public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attr) {
      System.out.printf("Begin Directory: %s%n", dir);
      return CONTINUE;
   }

   // Print information about each file/symlink visited.
   @Override
   public FileVisitResult visitFile(Path file, BasicFileAttributes attr) {
      if (attr.isSymbolicLink()) {
         System.out.printf("Symbolic link: %s ", file);
      } else if (attr.isRegularFile()) {
         System.out.printf("Regular file: %s ", file);
      } else {
         System.out.printf("Other: %s ", file);
      }
      System.out.println("(" + attr.size() + "bytes)");
      return CONTINUE;
   }

   // Print the directory visited.
   @Override
   public FileVisitResult postVisitDirectory(Path dir, IOException ex) {
      System.out.printf("End Directory: %s%n%n", dir);
      return CONTINUE;
   }

   // If there is an error accessing the file, print a message and continue.
   @Override
   public FileVisitResult visitFileFailed(Path file, IOException ex) {
      System.err.println(ex);
      return CONTINUE;  // or TERMINATE
   }

   // main
   public static void main(String[] args) {
      try {
         Path startingDir = Paths.get("..");
         Files.walkFileTree(startingDir, new WalkFileTreeTest());
      } catch (IOException ex) {
         System.out.println(ex);
      }
   }
}
```

The second version takes 2 additional arguments: the *`options`* specifies whether to follow symlink (e.g., `EnumSet.noneOf(FileVisitOption.class)` or `EnumSet.of(FileVisitOption.FOLLOW_LINKS)`); the *`maxDepth`* specifies the levels to visit (set to `Integer.MAX_VALUE` for all levels).

```
public static Path walkFileTree(
      Path startDir, Set<FileVisitOption> options, int maxDepth, FileVisitor<? super Path> visitor)
      throws IOException
```

For example, the following main() method can be used for the previous example.

```
   public static void main(String[] args) {
      try {
         Path startingDir = Paths.get("..");
```

```
         EnumSet<FileVisitOption> opts = EnumSet.of(FileVisitOption.FOLLOW_LINKS);
            // or EnumSet.noneOf(FileVisitOption.class)
         Files.walkFileTree(startingDir, opts, 2, new WalkFileTreeTest());
            // use Integer.MAX_VALUE for all level
      } catch (IOException ex) {
         System.out.println(ex);
      }
   }
```

10.9 Watching the Directory for File Changes - WatchService

[TODO]

LINK TO JAVA REFERENCES & RESOURCES

MORE REFERENCES & RESOURCES

1. Java Online Tutorial on "Basic I/O" @ http://download.oracle.com/javase/tutorial/essential/io/index.html, in particular "File I/O (Featureing NIO.2)".

Latest version tested: JDK 1.7.0_03
Last modified: May, 2012