CSE 11: Lecture 10

- ✔ File I/O
- ✔ Text files, binary files
- ✔ Important classes in the java.io package
- ✔ The containment pattern
- ✔ Text file output: PrintWriter, FileWriter
- ✔ Text file input: BufferedReader, FileReader
- ✔ StringTokenizer

(Reading: Savitch, Ch. 9)

File I/O

- So far:
  - Input to your programs has come from the keyboard as characters typed in
  - Output from your programs has gone to the terminal screen as characters displayed
- Other extremely useful forms of Input/Output are:
  - Graphical I/O, with a graphical user interface (GUI)
  - File I/O
- We will look at file I/O now, graphical I/O later
- The file I/O classes we will be talking about are in the `java.io` package...
  - You can put `import java.io.*;` at the top of your source code files to make referring to these classes easy.
  - (the `*` means "all the classes in the package")
- Using these classes involves understanding both inheritance (the IS-A relationship) and containment (the HAS-A relationship)

Binary files and text files

- Files are structures that hold data
- Like data in main memory, a file consists of a sequence of bytes
- Unlike data in memory, files are implemented using secondary storage (disk), and so data in files can persist even after your program ends
- How the sequence of bytes in a file is interpreted depends on what methods are used to access it
- Files can be classified into two kinds:
  - text files, that hold byte values that are meant to be interpreted as human-readable text (example: a .java file that you've created with your text editor)
  - binary files, that hold raw binary data whose values are not intended to be interpreted as human-readable text (example: a .class file that javac creates)
  - this is the only difference between text and binary files under Unix!
  - ... any text file can be treated as a binary file, but some binary files cannot meaningfully be treated as a text file

Text or binary?

- Suppose a file consists of this sequence of 4 bytes:

01001000	01101001	00001010	00100001

- Is this a text file or a binary file?

- In decimal, these binary byte values are 72, 105, 10, 33, which are the characters '`H`', '`i`', newline, and '`!`' in the ASCII coding system (see appendix 3 in Savitch for a partial list of these)

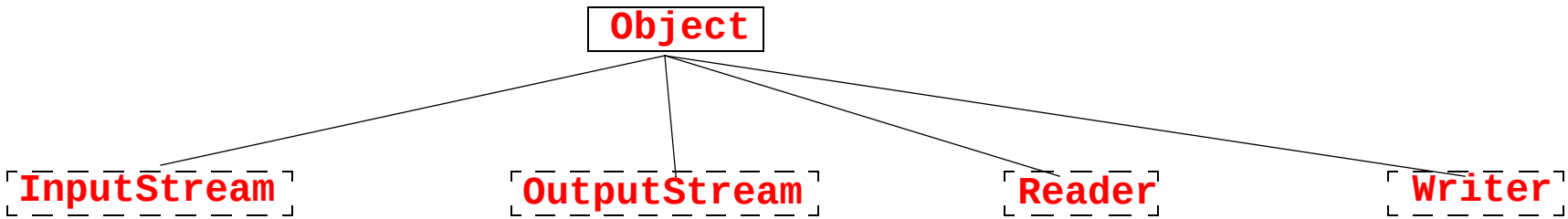
- If you looked at this file in a text editor, it would look like this (the newline character is not visible; it serves to help format the display):

```
Hi
!
```

- Since all the bytes in the file are ASCII characters, it could be considered a text file
- ...But it could also be considered a binary file! For example, those 4 bytes could be interpreted as a 32-bit binary int, representing decimal value 1,214,843,425

A tour of (some of) the java.io package

- The java.io package contains standard Java library classes used for I/O
- We will look briefly at some of the most important classes in that package, to understand how and why to use them
- (For more information, see the online documentation for the java.io package)
- The main "top-level" classes in this package are InputStream, OutputStream, Reader, and Writer
- These are all *abstract* classes (you cannot create an object of an abstract class; it is used to define a type with methods, which will be a superclass for other classes)
- These are sublcasses of Object, and in turn have subclasses, which we'll look at soon



- ... First, let's look at these classes

InputStream, OutputStream, Reader, Writer

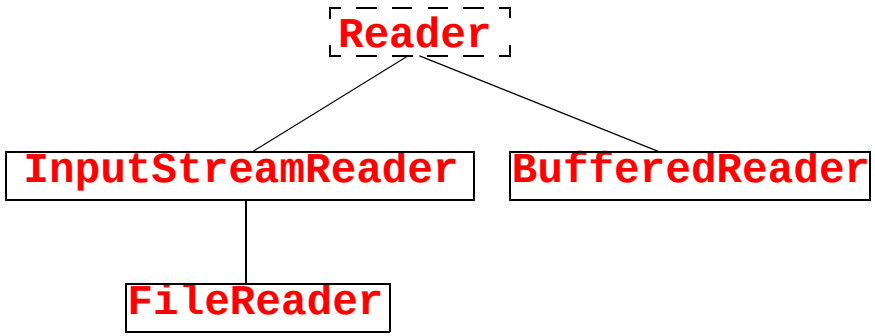
- As their names suggest, InputStream and Reader objects are used to read input from some source
  - The source can be a keyboard, or a file, or a Unix pipe from another process, or a socket connection to another computer on the net, etc., etc.
- And as their names suggest, OutputStream and Writer objects are used to write output to some destination
  - The destination can be a terminal screen, or a file, or a Unix pipe to another process, or a socket connection to another computer on the net, etc.
- An important difference:
  - Reader and Writer objects are for character-oriented I/O (“text”)
  - InputStream and OutputStream objects are for byte I/O (“binary”)

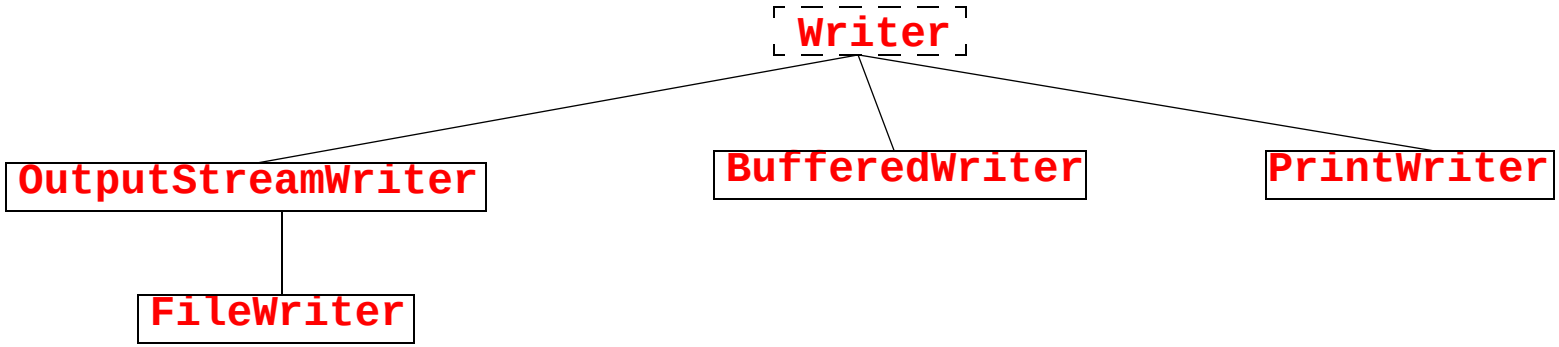
Reader and Writer

- Reader and Writer are abstract classes, so you can't create any Reader or Writer objects directly
  - However, they have many nonabstract subclasses; you can create objects of these subclasses
- Objects that are instances of subclasses of Reader and Writer should be used when you want to do character-oriented , i.e., text I/O
  - In Java, the char data type is 16 bits, and can represent any Unicode character
  - In Java, String objects consist of a sequence of chars
  - So if you want to do I/O involving text, i.e. Strings or single chars possibly organized into lines of text, use Reader (for input) or Writer (for output) classes

Some useful subclasses of Reader and Writer

- Some useful subclasses of Reader are InputStreamReader, FileReader, and BufferedReader
- Some useful subclasses of Writer are OutputStreamWriter, FileWriter, PrintWriter, and BufferedWriter





Reader methods

- The fundamental method introduced in the Reader class is the read() method, declared this way:

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

- This method does a read operation from an input source, and
  - returns a char as the least significant 16 bits of an int value
  - returns -1 if the read operation failed because end-of-input was reached
  - or, throws an IOException if there was some problem with the operation
- Subclasses of Reader will override the read() method in particular ways:
  - For example, FileReader defines read() to read one or more bytes from a file, and convert it to a Unicode char according to a “localization” scheme (ASCII coding is the default)
- Subclasses of Reader will also provide additional methods that may be useful:
  - For example, BufferedReader introduces the readLine() method, which reads an entire line of input (a sequence of characters terminated by e.g. a linefeed character) and returns it as a String (not including the line terminator)

Reader methods, cont'd

- The Reader classes also provide the close() method, declared this way:

```
public void close() throws IOException {
```

- This method closes the connection to the input source

- Subclasses of Reader will define close() appropriately; for example, FileReader's close() method will close the file associated with the Reader

- How to open a connection to an input source?
  - ... by creating an instance of an appropriate Reader class using an appropriate constructor

Writer methods

- The fundamental method introduced in the Writer class is the write() method, declared this way:

```
public void write(int c) throws IOException {
```

- This method does a write operation to an output destination
  - it takes an int argument, and writes the character value stored in the least-significant 16 bits of it
  - or, throws an IOException if there was some problem with the operation
- Subclasses of Writer will override the write() method in particular ways:
  - For example, FileWriter defines write() to write a single byte to a file, converting from a Unicode character according to a “localization” scheme (ASCII coding is the default)
- Subclasses of Writer will also provide additional methods that may be useful:
  - For example, PrintWriter introduces print() and println() methods, which write primitive types as Strings optionally terminated by a line terminator (e.g. a linefeed, newline, or linefeed and newline)

Writer methods, cont'd

- The Writer classes also provide the flush() method, declared this way:

```
public void flush() throws IOException {
```

- This method "flushes" the connection to the output destination, i.e. it sends all characters that may be waiting to be sent to the destination

- The Writer classes also provide the close() method, declared this way:

```
public void close() throws IOException {
```

- This method first calls the flush() method, and then closes the connection to the output destination

- Subclasses of Writer will define flush() and close() appropriately

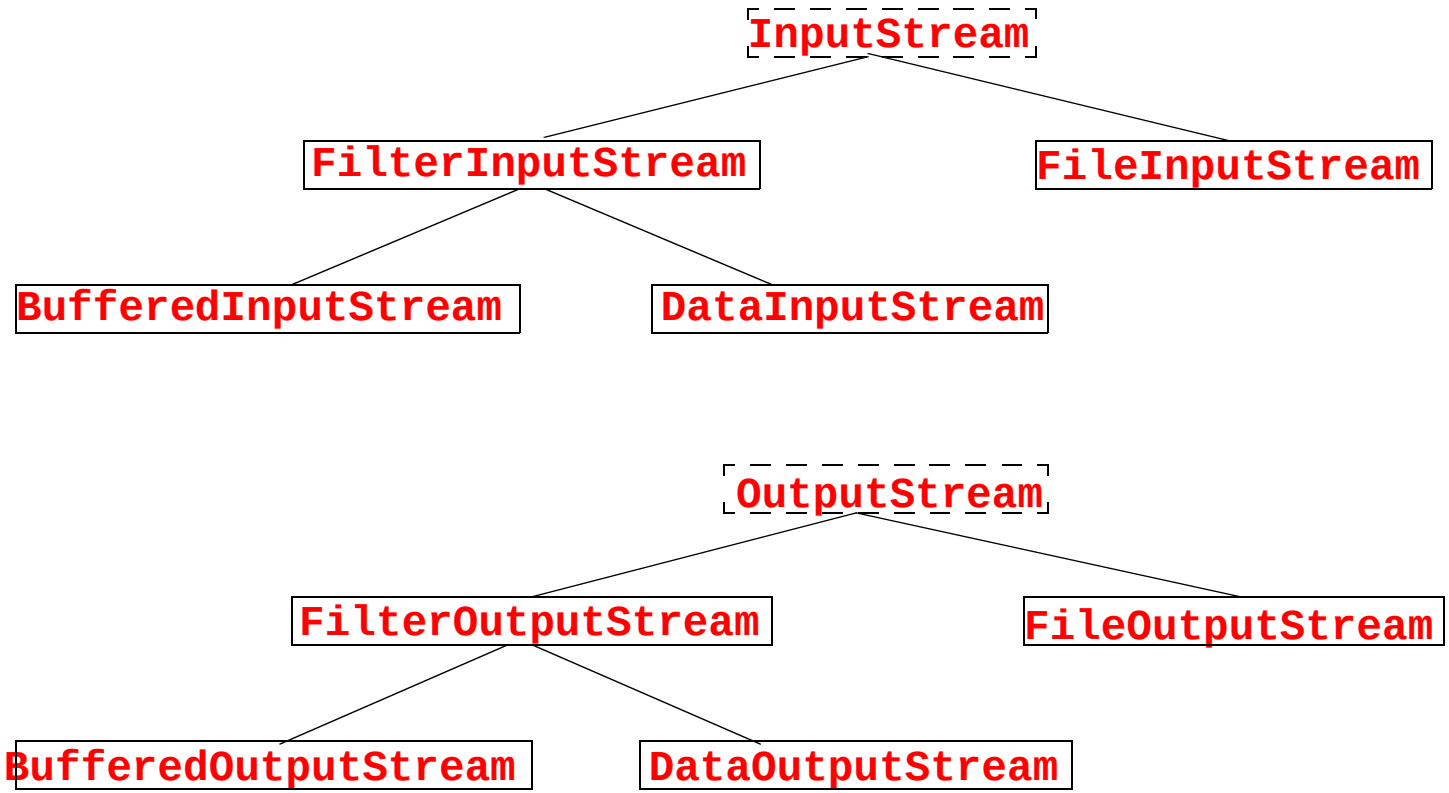
- How to open a connection to an output destination?
  - ... by calling an appropriate constructor of an appropriate Writer subclass

InputStream and OutputStream

- ✔ InputStream and OutputStream are abstract classes, so you can't create any InputStream or OutputStream objects directly
- ✔ However, they have many nonabstract subclasses; you can create objects of these subclasses
- ✔ Subclasses of InputStream and OutputStream should be used when you want to do binary I/O
  - ✗ With binary I/O, there is no conversion to, or from, Unicode chars based on a "localization" scheme
  - ✗ With binary I/O, there is no assumption that the input or output will be organized into lines of characters terminated with a newline, etc., that it will be viewable in a text editor, etc.
  - ✗ So if you want to do I/O that is not character or line oriented, use InputStream (for input) or OutputStream (for output) classes

Some useful subclasses of InputStream and OutputStream

- Some useful subclasses of InputStream are FileInputStream, BufferedInputStream, DataInputStream
- Some useful subclasses of OutputStream are FileOutputStream, BufferedOutputStream, DataOutputStream



InputStream methods

- The fundamental method of all InputStream classes is the read() method, declared this way:

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

- This method reads a single byte from an input source, and
  - returns the byte as the least significant 8 bits of an int value
  - returns -1 if the read operation failed because end-of-input was reached
  - or, throws an IOException if there was some problem with the operation
- Subclasses of InputStream will override the read() method in particular ways:
  - For example, FileInputStream defines read() to read a single byte from a file
- Subclasses of InputStream will also provide additional methods that may be useful:
  - For example, DataInputStream introduces methods for reading binary-format primitive types: readInt(), readChar(), readDouble(), etc., and returning the value read

InputStream methods, cont'd

- The InputStream classes also provide the close() method, declared this way:

```
public void close() throws IOException {
```

- This method closes the connection to the input stream, and releases any system resources associated with the stream.

- Subclasses of InputStream will define close() appropriately; for example, FileInputStream's close() method will close the file associated with the stream

- How to open a connection to an input source?
  - ... by calling an appropriate constructor of an appropriate InputStream subclass

OutputStream methods

- The fundamental method of all OutputStream classes is the write() method, declared this way:

```
public void write(int b) throws IOException {
```

- This method does a write operation to an output destination
  - it takes an int argument, and writes the byte value stored in the least-significant 8 bits of it
  - or, throws an IOException if there was some problem with the operation
- Subclasses of OutputStream will override the write() method in particular ways:
  - For example, FileOutputStream defines write() to write a single byte to a file
- Subclasses of OutputStream will also provide additional methods that may be useful:
  - For example, DataOutputStream introduces methods for writing portable binary-format primitive types: writeInt(int v), writeChar(int c), writeDouble(double d), etc.

OutputStream methods, cont'd

- Like the Writer classes, OutputStream classes define flush() and close() methods

Containment and the java.io classes

- How do you actually do input from some data source, or output to some data destination, with the desired text or binary I/O functionality?
- The design of the java.io classes usually makes this a three-step process:
  - 1. Select a class that is designed to connect to the kind of data source/destination you want; create an instance of this class using a constructor and argument that will initialize the desired connection
  - 2. Select a class that has the kind of I/O functionality you want; create an instance of this class, using a constructor that takes as argument the object created in step1 (this makes it one of the new object's instance variables)
  - 3. Use methods of the object created in step 2 to do the I/O. These methods will call methods of the contained object to do the I/O, plus do some additional work as needed

- This is an example of the "containment" or "composition" pattern in object-oriented design: an object of one class X contains, or is composed of, an object of a class Y
- With containment, you can say every X has-a Y. (This is different from inheritance, where every X is-a Y!)
- This is sometimes also called "delegation", because X's methods delegate some of their work to methods of the contained Y object

Doing text file output

- Following the containment pattern to do line-oriented text file output to a file named `myfile` we can proceed as follows:
  - 1. Select a class that is designed for text output to files: for example, `FileWriter`. Create an instance of this class using a constructor and argument that will initialize the connection:

```
FileWriter fw = new FileWriter("myfile");
```

  - 2. Select a class that has a println() method: for example, `PrintWriter`. Create an instance of this class, using a constructor that takes as argument the object created in step1:

```
PrintWriter pw = new PrintWriter(fw);
```

  - 3. Use methods of the object created in step 2 to do the I/O:

```
pw.println("The answer is: " + 2 + "+" + 2 + "=" 2 + 2);
```

- Often you combine steps 1 and 2 in one statement:

```
PrintWriter pw = new PrintWriter (new FileWriter("myfile"));
```

- You are used to doing text console output in Java using methods of the `System.out` object, which is an instance of `java.io.PrintStream`; `java.io.PrintWriter` is very similar to `PrintStream`

FileWriter constructors

- You create a **`FileWriter`** object associated with a file by passing the **`FileWriter`** constructor the name of the file, as a String.

- There are two FileWriter constructors that are commonly used:

```
new FileWriter("myfile")
```

... this will create an empty file named **`myfile`** in the current working directory if it doesn't already exist, and if it does exist, it will truncate it, i.e. *make* it empty! (So watch out.) Then it creates a FileWriter object associated with the file, ready for output.

```
new FileWriter("myfile", true)
```

... this will create an empty file named myfile if it doesn't already exist, and if it does exist, it will leave it alone. (So output to the file will be appended, i.e. added at the end.) Then it creates a FileWriter object associated with the file, ready for output.

Text file output: Some additional points

- When you are done with output to the file, you should close the PrintWriter by calling its close() method:

  ```
  pw.close();
  ```

  This will help to ensure that you don't lose data.

- The FileWriter constructors throw `IOException` objects! For example, if you don't have write permission for the file you named. You should wrap the call in a try-catch block and handle it appropriately

- If the String filename argument can be the full pathname of a file (it does not have to be relative to the current working directory)

Using PrintWriter for text output

- Consider this example:

```
import java.io.*;
public class Test {
   public static void main(String args[]) throws IOException {
      PrintWriter pw = new PrintWriter(
            new FileWriter("foo.txt"));
      pw.println("Hi");
      pw.print('!');
      pw.close();
   }
}
```

- After this runs, the file foo.txt contains 4 bytes whose values are exactly what is shown on page 4 of this lecture! When viewed in a text editor, it would look like

```
Hi
!
```

Using PrintWriter for text output, cont'd

- Consider this different example:

```
import java.io.*;
public class Test {
   public static void main(String args[]) throws IOException {
      PrintWriter pw = new PrintWriter(
            new FileWriter("foo.txt"));
      pw.print( 72 );  pw.print(  105 );
      pw.print(  10 );  pw.print(  33 );
      pw.close();
   }
}
```

- After this runs, the file foo.txt contains 9 bytes, and if viewed in a text editor, would look like:

721051033

- .... the same would result from `pw.print( 721051033 );`

Input from a text file

- Getting input from a text file is similar to getting input from the keyboard
- To do input from the keyboard, you receive a sequence of bytes
  - these bytes are interpreted as text characters that have been typed
  - some bytes correspond to visible characters, some do not (such as space, newline, tab, backspace, etc.) but they are all text characters

- We have been using methods of the SavitchIn class for keyboard input
- As we saw, SavitchIn builds its methods "on top of" `System.in.read()`, which reads a byte at a time from the keyboard input
- `System.in` is an instance of the `InputStream` class...
- For text file input we'll be using the `BufferedReader` class, which also has a `read()` method.
- You could duplicate all the SavitchIn methods using this, instead of `System.in.read()`, and read a character at a time from the file...
- Or, you can use the `readLine()` method of `BufferedReader`, which returns an entire line as a String

Doing text file input

- Following the containment pattern, to do line-oriented text file input from a file named `myfile` we can proceed as follows:
  - 1. Select a class that is designed for text input from files: for example, `FileReader`. Create an instance of this class using a constructor and argument that will initialize the connection:

```
FileReader fr = new FileReader("myfile");
```

  - 2. Select a class that has a readLine() method: for example, `BufferedReader`. Create an instance of this class, using a constructor that takes as argument the object created in step1:

```
BufferedReader br = new BufferedReader(fr);
```

  - 3. Use methods of the object created in step 2 to do the I/O:

```
String line = br.readLine();
```

- Often you combine steps 1 and 2 in one statement:

```
BufferedReader br = new BufferedReader (new FileReader("myfile"));
```

FileReader constructor, and exceptions

- ✔ You create a **`FileReader`** object associated with a file by passing the **`FileReader`** constructor the name of the file, as a String:

```
new FileReader("myfile")
```

... this will create a FileReader object connected to a file named **`myfile`** in the current working directory, and prepare the file for reading, starting at the beginning of the file.

  - ✗ If the file does not already exist, or you do not have permission to read it, a **`FileNotFoundException`** will be thrown. These must be caught or declared

- ✔ The read() and readLine() methods of BufferedReader throw **`IOException`** objects. These must be caught or declared

End-of-file

- When reading input from a file, you usually want to start at the beginning of the file, and go on to the end
- The technique we have seen for creating a BufferedReader object associated with a file will ensure that reading starts at the beginning...
- ... but how can you tell when you have reached the end of the file and there is nothing more to be read?
  - Note that this is very different from reading the ASCII end-of-file character from the file, which is just another byte value (decimal 4)!
- Classes in the java.io package use different ways of telling you when this has happened:
  - some methods throw a **`EOFException`** to signal this (we'll see this when we talk about binary file I/O)
  - some methods return a distinctive value to signal this, that can't be confused with any actual data in the file (the read() and readLine() methods of BufferedReader are like this)
- We will look at the documentation for the read() and readLine() methods of BufferedReader, and then write some code that uses them to do something useful

readLine(), in BufferedReader.java

```
/**
 * This method reads a line of text. Lines are terminated by
 * "\n", "\r", or "\r\n". The line terminators are not returned
 * with the line string.
 * @return  A String containing the line just read, or
 *          null if the end of the stream has been reached.
 * @exception IOException   If any kind of I/O error occurs.
 */
public String readLine() throws IOException
```

read(), in BufferedReader.java

```
/**
 * This method returns the next character from the buffer. If all
 * the characters in the buffer have been read, the buffer is
 * filled from the underlying Reader, and the next character is
 * returned. If the buffer does not need to be filled, this method
 * returns immediately. If the buffer needs to be filled,
 * this method blocks until data is available from the underlying
 * Reader, the end of the  stream is reached, or an exception is
 * thrown.
 * @return   The next character of data as an int, or
 *           -1 if the end of the stream is encountered.
 * @exception IOException   If any kind of I/O error occurs.
 */
public int read() throws IOException
```

Copying from one text file to another, a line at a time

```
import java.io.*;
public class CopyV1 {
   public static void main( String[] args ) throws IOException {
      System.out.print("Enter input file name: ");
      String inFileName = SavitchIn.readLine();
      System.out.print("Enter output file name: ");
      String outFileName = SavitchIn.readLine();

      BufferedReader br = new BufferedReader(
              new FileReader(inFileName));
      PrintWriter pw = new PrintWriter(
              new FileWriter(outFileName));

      String line;
      while ( (line = br.readLine()) != null ) {  // catch EOF
         pw.println(line);
      }
      pw.close();
   }
}
```

Copying from one text file to another, a character at a time

```
import java.io.*;
public class CopyV2 {
   public static void main( String[] args ) throws IOException {
      System.out.print("Enter input file name: ");
      String inFileName = SavitchIn.readLine();
      System.out.print("Enter output file name: ");
      String outFileName = SavitchIn.readLine();

      BufferedReader br = new BufferedReader(
              new FileReader(inFileName));
      PrintWriter pw = new PrintWriter(
              new FileWriter(outFileName));

      int ch;
      while ( (ch = br.read()) != -1 ) {   // catch EOF
         pw.print((char)ch);
      }
      pw.close();
   }
}
```

The StringTokenizer class

- In the java.io classes that deal with text input, there are methods like
  - read() for reading a single character, or
  - readLine() for reading an entire line of input as a String
- There are not any methods for reading an integer, or a double, or a whitespace-delimited word from a text stream (that's why Savitch wrote SavitchIn)
- But there is a class in the `java.util` package that provides a handy way of breaking up a String into smaller pieces ("tokens") for further processing: the `StringTokenizer` class

The StringTokenizer class, cont'd

- Create an instance of StringTokenizer by starting with a String:

```
StringTokenizer tok = new StringTokenizer("Here's a string...");
```

- Now use instance methods of the StringTokenizer to get whitespace-delimited words ("tokens") from the String, one at at time. Here are the method headers:

```
// return the next token available in the StringTokenizer
public String nextToken()

// return true if there are more tokens available in the
// StringTokenizer, else return false
public boolean hasMoreTokens()

// return the number of tokens available in the StringTokenizer
public int countTokens()
```

- There is also a constructor that lets you specify both a String to be tokenized, and a string of *delimiter characters* to use in tokenizing

```
new StringTokenizer("Here's a string...", "'.")
```

- The default delimiter character string is `" \t\n\r"` (that is, space, tab, newline, carriage return)

Using StringTokenizer: an example

```
StringTokenizer tok = new StringTokenizer("Here's a string...?");
while(tok.hasMoreTokens()) {
   System.out.println(tok.nextToken());
}
```

- prints..

```
Here's
a
string...?
```

```
StringTokenizer tok =
   new StringTokenizer("Here's a string...?", "'.");
while(tok.hasMoreTokens()) {
   System.out.println(tok.nextToken());
}
```

- prints...

```
Here
s a string
?
```

Summary: basic file I/O in Java

- Text output to a file: wrap a `FileWriter` object in a `PrintWriter` object, and call the `PrintWriter`'s methods to do output.
- Text input from a file: wrap a `FileReader` object in a `BufferedReader` object, and call the `BufferedReader`'s methods to do input.
- Next time we will discuss binary I/O methods:
- Primitive type binary output to a file: wrap a `FileOutputStream` object in a `DataOutputStream` object, and call the `DataOutputStream`'s methods to do output.
- Primitive type binary input from a file: wrap a `FileInputStream` object in a `DataInputStream` object, and call the `DataInputStream`'s methods to do input.

Next time...

- ✔ I/O streams reviewed
- ✔ Binary file I/O: DataInputStream, DataOutputStream
- ✔ Buffering
- ✔ The File class
- ✔ GUI programming, Swing, JFC, and the AWT
- ✔ Components and Containers
- ✔ JPanel and Graphics objects
- ✔ JLabels, JButtons, and JTextFields

(Reading: Savitch, Ch. 9 and Ch. 12)