How to Make Double-Clickable Java Applications
and other things beginning Java students ask about that there is usually not enough time to explain

Allen Stix
Mary F. Courtney
This is the back to school issue of Technical Reports. It is dedicated to all students aspiring to be Java aficionados.

The authors hope that it makes amends for dodging these questions when answers would have taken up too much time in class.

*Allen Stix* is Associate Professor in the Computer Science Department at Pace University, based in Westchester.

*Mary F. Courtney* is Professor of Computer Science at Pace University, based in Westchester, where she has been teaching courses in programming, algorithms, and graphics for over 25 years.

Professor Courtney holds a B.A. in Mathematics from the College of Mount Saint Vincent, an M.A. in Mathematics from Lehman College, an M.S. in Computer Science from Pace University, and an Ed.D. in the College Teaching of Mathematics with an emphasis in Computer Science from Teachers College, Columbia University.
How to Make Double-Clickable Java Applications
and other things beginning Java students ask about that there is usually not enough time to explain

Allen Stix and Mary F. Courtney

Table of Contents

1) Holding the Action—How to program a processing pause  page 2
2) Timing the execution of a section of code – How to read the computer’s clock  page 5
3) Automatic code documentation with the C:\> javadoc command  page 6
4) Packages – How to place, run, and compile applications that reside in a package  page 9
5) Bundling a collection of files as a Java Archive (JAR) file  page 17
6) I/O between your computer and another computer  page 27
7) Writing an applet so that it opens both as an applet and as an application  page 35

Appendix: Notes on the CD (Change Directory) and MD (Make Directory) commands  page 37

0) Introduction

When computer science faculty members speak of learning to program, they are thinking of developing creative abilities for solving new problems and becoming familiar with symbolic constructs promising long-term utility. Little in the present paper helps toward these ends. For instance, once you know how to make one application double-clickable (this is a use of JAR files), you can make any application double clickable with absolutely no new thinking. This process would take a good chunk of class time to illustrate, yet when Java is superseded, it will have no further applicability. Reputable computer science instructors tend to avoid dead-end topics like these.

Yet, the technology exemplified here is often asked about by inquisitive students in Programming I and II. They become frustrated when effective answers to these legitimate questions are difficult to come by. We offer this as a "missing manual." If the techniques herein add fun to programming and fun translates into time on task, this is where these superfluous topics pick-up a modicum of value.

One disclaimer and then we'll get underway: The explanations here are deliberately abbreviated. In each case, the intent is to provide a foothold. Space does not permit a full blown background or an exhaustive treatment of the associated features, frills, wrinkles, tricks, and traps.
1) Holding the Action—How to program a processing pause

A loop prints your name ten times, or a line of sixty asterisks. An inquisitive student asks how to delay the action between iterations so that the output appears one name at a time, or one asterisk at a time; not all at once as a fait accompli.

The statement that causes a suspension of processing is:

```java
Thread.sleep(durationInMilliseconds);
```

an integer value up to `Long.MAX_VALUE` (which is 9223372036854775807) giving the length of the pause in milliseconds

The statement `Thread.sleep(500);` causes a 500 millisecond delay. A millisecond is a thousandth of a second, so 500 milliseconds is \( \frac{500}{1000} = \frac{1}{2} \) a second.

`Thread.sleep()`, however, needs scaffolding to work. The recommended approach is to use a `try/catch` structure as shown below:

```java
try {
    Thread.sleep(1000);
}
catch(Exception e) {
}
```

The alternative is to put a `throws` clause onto the heading of the method containing `Thread.sleep()`, such as:

```java
public static void main(String[] args) throws Exception {
    //details omitted
    Thread.sleep(10); //no try/catch needed
    //details omitted
}
```
The complication associated with using a `throws` clause is exception propagation. Suppose we had the `three()` method shown below and that it was running:

```java
void three() throws Exception
{
    //details omitted
    Thread.sleep(10); //no try/catch needed
    //details omitted
}
```

Suppose, also, that the `main()` method had called method `one()`, `one()` had called `two()`, and `two()` had called `three()`. With no `try/catch` structure in the code to handle `Thread.sleep()`'s potential for throwing a checked exception, Java requires a `throws` clause on each method in the calling sequence; which is `three()`, `two()`, `one()`, and `main()`. This can be a nuisance when methods that had been working fine suddenly require maintenance because of a new method or the addition of "a pause" someplace else in the system. That's why nesting `Thread.sleep()` in a `try/catch` is the better alternative.

The following two programs do the same thing; they print a line of sixty asterisks. The one on the left includes the `try/catch` phrase. The one on the right does not (accordingly, the `main()` method requires a `throws` clause). The exercise is to change the asterisk display rate.

```java
class Asterisks1
{
    public static void main(String[] args)
    {
        for(int i = 0; i < 60; i++)
        {
            System.out.print("*");
            try
            {
                Thread.sleep(100);
            }
            catch(Exception e)
            {
            }
        }
        System.out.println();
    }
}
```

```java
class Asterisks2
{
    public static void main(String[] args) throws Exception
    {
        for(int i = 0; i < 60; i++)
        {
            System.out.print("*");
            Thread.sleep(100);
        }
        System.out.println();
    }
}
```

A good technique, especially if you want to insert pauses at many points in a program, is to write utility such as the following. You can call it wherever you like, as often as you like, without ever needing to code either a `try/catch` or a `throws` clause.

```java
class Pause
{
    static void ms(long duration_in_milliseconds)
    {
        try
        {
            Thread.sleep(duration_in_milliseconds);
        }
        catch
        {
        }
    }
}
```
Here is how you'd use it:

class Asterisks3
{
    public static void main(String[] args)
    {
        for(int i = 0; i < 60; i++)
        {
            System.out.print("**");
            Pause.ms(100);
        }
        System.out.println();
    }
}
2) Timing the execution of a section of code – How to read the computer's clock

The clock ticks in milliseconds, and the time reads out in milliseconds since midnight, Greenwich Mean Time, at the start of January 1, 1970. The method that reads the clock is in the `System` class, returns a `long`, and has the signature:

```
public static long currentTimeMillis()
```

Variables of type `long` are signed, 64-bit integers, therefore the number of milliseconds will not overflow until the year 292,289,995.

The code below illustrates measuring the number of millisecond required by the `Arrays.sort()` method to sort the `data` array:

```
long startTime = System.currentTimeMillis();
java.util.Arrays.sort(data);  //the method being timed
long stopTime = System.currentTimeMillis();
long duration = stopTime - startTime;
```

Because time is a number that steadily increases, no provision needs to be made for crossing from a.m. to p.m. or from one day to the next.

The only concern is Java's garbage collection facility which is activated automatically, as needed, to release the memory of discarded (i.e. unreachable) objects. We don't want it to kick-in while we're making a timing, especially if our intent is to compare the performance of one algorithm or implementation with another. Protection is acquired by explicitly calling for garbage collection with `System.gc()` immediately beforehand, as shown below:

```
System.gc();  //activates garbage collection
long startTime = System.currentTimeMillis();
java.util.Arrays.sort(data);  //the method being timed
long stopTime = System.currentTimeMillis();
long duration = stopTime - startTime;
```

Calling `System.gc()` does not switch-off garbage collection, but it obviates its need to run in the near future. This is what you want, to "clear the 'memory leakage' state." If you are timing a section of code that does not create an undue heap of spent objects, your reading will not be affected by an untoward garbage collection. However, if you are timing a lengthy process involving significant amount of memory from the heap, you'd like a realistic appraisal of performance.
3) Automatic code documentation with the C:\> javadoc command

The invocation of the javadoc program in the title of this section, directly above, is misleading. This program may be started from any directory. The active directory does not need to be C:\. That said, let's move on.

Exactly the same kind of methodically organized .html files that Sun Microsystems supplies to document Java can be generated for your classes with the Javadoc tool that comes with your installation of the Java 2 Standard Edition. (Sun's documentation is reachable through http://java.sun.com/reference/api/index.html. When you Google the name of a class or a package such as StringTokenizer, JFrame, or java.util one of the first links is always directly into Sun's javadocs.)

The impressive, industrial-caliber documentation the Javadoc tool puts just keystrokes away is motivation to use good mnemonics and add crisp, clear comments.

To exemplify, below is the source code for the Box class, stored in the file E:\JavaPrograms\Box.java. The unfamiliar elements pertaining to Javadoc are:

- the Javadoc comment blocks:
  The first line of the comment block opens a slash followed by two asterisks: /**
  Each successive line starts with an asterisk at the left end:
  The block is terminated by a line that is blank apart from its leading asterisk and slash: */
  Note the alignment.

- the HTML (or XHTML) tags within Javadoc comment blocks for formatting control. (On the following page, the break tag is shown as <br>, making it an XHTML tag. In our browser, the HTML equivalent, <br>, would work just as well.)

- the special Javadoc tags, which are Javadoc keywords beginning with an @ sign

These unfamiliar elements are not required in order to use Javadoc. In other words, you can create Javadoc documentation whether or not the source code includes Javadoc comment blocks, whether or not the comment blocks incorporate HTML or XHTML formatting, and whether or not @ tags added. However, these are nifty features you'll surely want to use.
/**
 * This is a simple class to illustrate the creation of javadoc documentation.
 * Visualize a Box object as a rectangular swatch having
 * a symbol, a length, and a height. Here is a sample
 * 
 * "#####" <- Sample Box object
 * (symbol is '#', length is 9, height is 3)
 * 
 * @version 1.0, Created 09/30/07
 * @author Mr. Bunny (<u>Mr. Bunny's Big Cup o' Java</u> by Carlton Egremont III)
 * 
 */

public class Box
{
    private int length = 9;
    private int height = 4;
    private char symbol = '#';
    private int serialNumber;

    public static int numberCreated=0;

    public Box()
    {
        numberCreated = numberCreated + 1;
        serialNumber = numberCreated;
    }

    /**
     * Creates a new Box object with user specified dimensions,
     * increases the class variable that keeps track of the number
     * of Boxes created, and assigns the new Box a serialNumber
     * 
     * @param forLength The length of the new rectangle
     * @param forHeight The height of the new rectangle
     * @param forSymbol The character used to depict the Box
     */

    public Box (int forLength, int forHeight, char forSymbol)
    {
        length = forLength;
        height = forHeight;
        symbol = forSymbol;
        numberCreated = numberCreated + 1;
        serialNumber = numberCreated;
    }

    public String toString()
    {
        String s = "Box ": ";
        s += "length = " + length + ", ";
        s += "height = " + height + ", ";
        s += "symbol = " + symbol + "\n";

        return s;
    }
}

The Javadoc comment block that pertains to the class is placed ahead of the class.

The Javadoc comment block that pertains to a method is placed ahead of the method.
The **Javadoc** comments that introduce a class are placed in the source code above the class. The **Javadoc** comments that pertain to a method are placed directly ahead of the method. The **Javadoc** comments about a variable immediately precede the variable. Documenting any particular thing does not obligate you to document any other. You'll see how **Javadoc** comments appear in the documentation when you run `javadoc` and view the generated *index.html* file (and the other *.html* files). One more thing. HTML tags are never required by the **Javadoc** facility, but certain HTML tags are allowed within **Javadoc** comments to provide formatting control. For example, `<br>` (or `<br/>`) inside a "doc comment" will cause a line feed. The opening and closing `<u>` and `</u>` tags will supply underlining. Surrounding a Java keyword, the name of a class, the name of a method, or a fragment of code by `<code>` and `</code>` provides a special fixed-pitch font. This is commonly done. A fragment of text may be enclosed between `<font color=red>` and `</font>` tags to specify its color (red is just for illustration).

To generate the **Javadoc** documentation for the source code of the **Box** class in the file `Box.java`, which is in the folder `E:\JavaPrograms`, we would use the command:

```
E:\JavaPrograms> javadoc -author -version -private Box.java
```

To see the documentation, use your browser to view the file *index.html* that was written into the same folder as the source code. Look at `Box.html` and the other *.html* files as well.

The **Javadoc** documentation could have been created with the same command but without the three command line options `-author`, `-version`, and `-private`. These are individual switches. `-author` switches on the display (in the generated documentation) of the author(s) indicated in the source code at the `@author` tag. `-version` does similarly for the `@version` tag. `-private` includes `private` members (instance variables, class variables, and methods) of the **Box** class in the generated documentation. Without these, the generated documentation would not display the author, the version, or the instance variables specified as `private`. We used `@param` tags to evoke the stylized explanation of the parameters for the three argument constructor.

The information above will get you off to a good start. For more information about **Javadoc** and its features, see "How to Write Doc Comments for the Javadoc Tool" at Sun's site: [http://java.sun.com/j2se/javadoc/writingsdoccomments](http://java.sun.com/j2se/javadoc/writingsdoccomments), the **Javadoc** Tool's home page at Sun's [http://java.sun.com/j2se/javadoc](http://java.sun.com/j2se/javadoc), and Wikipedia's discussion at [http://en.wikipedia.org/wiki/Javadoc](http://en.wikipedia.org/wiki/Javadoc). Java in a Nutshell by David Flanagan and other reference books and textbooks also provide information about the **javadoc** program and **Javadoc** conventions.

**Warning**: Had not have experienced this we would have trouble believing it.

Certain computers on which Java compiles and runs perfectly will not create **Javadoc** comments. Running the `javadoc` program results in a `java.lang.IllegalArgumentException` and a screenful of reports on its propagation.

If this happens to you, run the `javadoc` program on another machine. And if you find out what's responsible for this problem, please share the information with us!
4) Packages – How to place, run, and compile applications that reside in a package

If your chief interest is in how to compile and run an application residing within a package, here is what to do. For the sake of illustration, we shall compile and run the hello world program, Hello.java, which is in the package named greetings. The first letter of the name of a package, by convention, is lower case. Package names in Java are case sensitive. This including each entry in the path to their folder, specifically the device (e.g. C:, or the the letter designating your flash drive).

Firstly, for the class (or classes) within a file of Java source code to be within a named package, the file must open with the **package** statement. The package statement, when present, must be the first statement in the file. Only comments may precede it; **import** statements, when present, have to follow it. You'll see the **package** statement below. Notice that it ends with a semicolon. Apart from this, the code is no different from the "hello world" program shown to students on their first day of Java, before packages became a consideration.

```java
package greetings;

public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello Packages");
    }
}
```

Secondly, files declared to be within a package belong inside a folder with the same name as the package. Thus, do not save Hello.java any old place, save it in a folder named greetings. This folder, however, may be located anywhere. Suppose the Hello.java file is saved on a flash drive as shown below. The flash drive is device E: on the present machine.

```
classpath
E:\JavaPrograms\PackageSamples\greetings\Hello.java
```

Remember that a folder and a directory are one and the same thing. A tutorial on how to make a new directory at the command prompt with the **md** command and on how to change the active directory at the command prompt with the **cd** command is in an appendix on pages 37-38.
Working directly with the Java Software Development Kit (i.e. with the SDK at the Command Prompt), one way to compile **Hello.java** is to make the **greetings** folder the active directory and compile as shown:

```
E:\JavaPrograms\PackageSamples\greetings> javac Hello.java
```

Alternatively, with any device and path as the active directory, the **Hello.java** file may be compiled by preceding the name of the file with its the full path:

```
C:\games> javac E:\JavaPrograms\PackageSamples\greetings\Hello.java
```

Both compilations shown place the created **Hello.class** file into the same folder holding that stored the source code, **greetings**, which is where it must be in order to be run. Thus, both of the shown above compilations create:

```
E:\JavaPrograms\PackageSamples\greetings\Hello.class
```

To run **Hello.class**, the following command may be used from any active directory on any device:

```
C:\>java -cp E:\JavaPrograms\PackageSamples\greetings\Hello.class
```
The purpose of packages is to organize software facilities. There are on the order of three thousand public classes in Java (the Java 2 Standard Edition). For the sake of organization, these are grouped into packages, of which there are around 150. If you build a software system, you'd want to put its classes, and whatever else you'd want to go with it such as image and sound files, into a separate package.

The classes that are first met by Java students (e.g. System, String, Math, Integer) are in the java.lang package. This package is implicitly imported into all Java source code files, automatically making each of its classes accessible. The result is that students can use System.out.println() and declare String objects without any thought to importing the java.lang.System class and the java.lang.String class individually, or importing java.lang.* to take in every class within the java.lang package.

Classes residing in all packages other than java.lang need to be imported. The StringTokenizer class resides in the package named java.util. If a method within a class you are coding uses a StringTokenizer object, your file will need the import statement shown below in either version 1 or version 2:

```java
//file: MyMess.java - version 1
import java.util.StringTokenizer;
public class MyMess {
    //details omitted
}
```

```java
//file: MyMess.java - version 2
import java.util.*;
public class MyMess {
    //details omitted
}
```

If you do not import the StringTokenizer class individually, as shown on the left, or along with all classes in the java.util package, as shown on the right, the compiler will flag a "cannot find symbol" error upon encountering a statement that declares a StringTokenizer reference variable or creates a StringTokenizer object. The errant "symbol" is StringTokenizer, the class's name; and the problem is that it is not within the file's name space.

There is one exception to the rule about needing to import classes other than those in java.lang. Any class resident in the same package as a user's class need not be imported to be accessed. In other words, all classes within the same package are automatically within each other's name space. But there is a catch. The classes in the ad hoc folder, such as the greetings folder, need to be on the classpath in order to be found. You may append any classpath you wish to your computer system's classpath variable (reachable through CONTROL PANEL | SYSTEM | ADVANCED tab | ENVIRONMENT VARIABLES button). If you do this, remember that the updated classpath does not take effect in an open Command Prompt window; you need to close it and re-open it. The preferred approach, however, is to enter it on the command line with the -cp option as shown at the bottom of page 10 in running the greetings.Hello. Bear in mind that an explicitly entered classpath completely preempts the system's setting. Along the entered classpath, different paths are separated with semicolons; no spaces are allowed; the active directory is represented by a period ( . ); the parent directory of the active directory is represented by two periods ( .. ).

This is a pretty dense -- beginning to sound like one of those manuals we so adore (not). The following example makes the central point clear and easy to manage.
For this example, the `HelloAgain.java` program uses an object of type `Salutation` and a `static` method in a class named `Hey`. These three classes are all in the `greetings` package. Our mission is to compile and run `HelloAgain.java` (in fact, all three classes need to be compiled). Here are the classes:

```java
// File: HelloAgain.java in the greetings folder
package greetings;
public class HelloAgain
{
    public static void main(String[] args)
    {
        System.out.println("Hello Again");
        Salutation salutation = new Salutation();
        salutation.sayHello("Yo! Howdy! How the #@$! are you?");
        Hey.greatToSeeYou();
    }
}
```

```java
// File: Salutation.java in the greetings folder
package greetings;
public class Salutation
{
    public void sayHello(String s)
    {
        System.out.println(s);
    }
}
```

```java
// File: Hey.java in the greetings folder
package greetings;
public class Hey
{
    public static void greatToSeeYou()
    {
        System.out.println("Great to see You!");
    }
}
```
In order to compile HelloAgain.java, set the active directory to the greetings folder and use the following command. What's new on the command line is the -cp (classpath) option and the specification of the classpath to the package holding the classes used by HelloAgain (which are Salutation and Hey). In the preceding, the -cp option was used with java, not javac.

For HelloAgain.java to compile, the classpath to Salutation and Hey must be given even though these classes are in the same package as HelloAgain.

```
E:\JavaPrograms\greetings> javac -cp E:\JavaPrograms HelloAgain.java
```

Because the E:\JavaPrograms is the parent directory of the active directory, E:\JavaPrograms\greetings, another way to express this same command is:

```
E:\JavaPrograms\greetings> javac -cp .. HelloAgain.java
```

That the .java files needed by HelloAgain.java are compiled, if need be, when HelloAgain.java is compiled is known as implicit compilation.

Another option is to compile Salutation.java and Hey.java individually before compiling HelloAgain.java:

```
E:\JavaPrograms\greetings> javac Salutation.java
E:\JavaPrograms\greetings> javac Hey.java
E:\JavaPrograms\greetings> javac HelloAgain.java
```

With compilation complete, to run HelloAgain use a command with same structure as on page 10. Both of the following work equally well. Regardless of the active directory, the classpath to the greetings folder must be explicit.

```
E:\JavaPrograms\greetings>java -cp E:\JavaPrograms greetings>HelloAgain
C:\>java -cp E:\JavaPrograms greetings>HelloAgain
```
What about a file such as MyMess.java, on page 11, which does not include the package statement? It is put into the unnamed default package. All classes within the unnamed default package, and stored within the same folder, are automatically within each other's name space. This is why a CS1 student who has no awareness of packages can build a bunch of classes, store them in a bunch of files, and have the classes accessible to each other. Relative to the javac and java command lines, the classpath is the active directory, which is represented by a single dot in the system's classpath environmental variable. Relative to the java command line, no package name is prepended to the name of the .class file that starts the application.

A set of rules surrounds the creation and use of packages. One of the rules is that only classes declared public are importable. When the public access modifier is left off the definition of the class, that class may only supply service (e.g. objects of its type; its static methods) to classes inside the same package; that class is "package private." When a class is declared public, the file holding its source code must have the same name as the class, case preserved. This naming rule is what enables Java to find source code for implicit compilation.

Classes may be public or package private, and only public or package private. The access modifiers private and protected cannot be applied to a class. However, once you start using packages, you'll be giving thought to protected methods within public classes. A protected method is "public" relative to all classes within the package and "private" relative to all classes not within the package. This is the same as when a method is declared with no access modifier. The difference is that a protected method is accessible within methods of classes that extend the ad hoc class, regardless of the extending class's package.

Finally, packages may be nested, meaning that a package may contain not only files but other packages. This is good to know relative to options for organizing systems and important relative to the rule that importing a package does not import a contained package. When classes from a nested package are needed, the contained package must be imported explicitly; its classes may be imported by name, or its full collection of classes may be imported using the asterisk. Always remember that the name of a subpackage consists of the name of its parent package, a dot (that is, a period), and the name of the subpackage. The name of the parent package is not part of the subpackage's classpath; it is part of its name.

A Java-learner comes upon a subpackage when using a JButton or some other GUI control. The JFrame and the JButton are imported from javax.swing. The Container and the FlowLayout manager are imported from java.awt, and the ActionListener and the ActionEvent interface and ActionEvent class are imported from the java.awt.event package, which is a package within java.awt. The coding could look like this:

```java
import javax.swing.*;  // for JFrame and JButton
import java.awt.*;    // for Container (the JFrame's content pane) and FlowLayout
import java.awt.event.*;  // for ActionListener and ActionEvent

class MyFirstJButton
{
    public static void main(String[] args)
    {
        // details omitted
    }
}
```
To illustrate the use of packages with a life-sized example, we offer the following.

Here is the class that contains the `main()` method where the application begins. The class is named `Main`. It is in the package named `p1` which is in the folder `C:\holder1`. The `main()` method uses the `message()` method from `ClassTwo`, which is in package `p2`, as well as the `message()` method from `ClassThree`, in package `p3`. Being in different packages from `Main` (and in different packages from each other) both `ClassTwo` and `ClassThree` need to be imported.

```java
// File: C:\holder1\p1\Main.java

package p1; // The class declared here, Main, is in the package p1
import p2.ClassTwo;
import p3.ClassThree;

public class Main
{
    public static void main(String[] args)
    {
        ClassTwo two = new ClassTwo(); // ClassTwo comes from package p2
        ClassThree three = new ClassThree(); // ClassThree comes from package p3

        System.out.println("line 1");
        two.message();
        three.message();
    }
}
```

Below is `ClassTwo`, residing in a package, `p2`, which is in an entirely different folder, `C:\holder2`:

```java
// File: C:\holder2\p2\ClassTwo.java

package p2; // The class declared here, ClassTwo, is in the package p2

public class ClassTwo
{
    public void message()
    {
        System.out.println("line from ClassTwo");
    }
}
```
Below is **ClassThree**, residing in a package, **p3**, which is in yet a different folder, C:\**holder3**:

```
Folder:       C:\holder3
Package:      p3
Class:        ClassThree

//File: C:\holder3\p3\ClassThree.java
package p3;   //The class declared here, ClassThree, is in the package p3
public class ClassThree
{
    public void message()
    {
        System.out.println("line from ClassThree");
    }
}
```

The individual classes may be compiled one by one; or, thanks to implicit compilation, **ClassTwo** and **ClassThree** can be compiled automatically through the compilation of **Main**. This happens because **Main** references **ClassTwo** and **ClassThree**. It works because of Java's conventions for naming and storing files. The following will compile and run the program from any directory. Here, the active directory is C:\.

To compile:

```
C:\>javac -cp C:\holder2;C:\holder3
```

*the classpaths to both package p2 and package p3 are required*

```
C:\>javac C:\holder1\p1\Main.java
```

*the full path to the file that holds the main() method*

*a semicolon separates the two classpaths*

To run:

```
C:\>java -cp C:\holder1;C:\holder2;C:\holder3
```

*classpaths to all used packages*

```
C:\>java p1.Main
```

*name of the file holding the main() method prepended by its package's name*

*semicolons separate the classPaths*

**Crucial:** The explicit classpath on the command line preempts the classpath entered as the environmental default (i.e. in **CONTROL PANEL | SYSTEM PROPERTIES | ENVIRONMENT VARIABLES**). Had the **Main** class been in the active directory, and not in a package, the **classpaths to all used packages** would have needed the dot (a period) to represent the active directory.
5) Bundling a collection of files as a **Java Archive (JAR) file**

A **JAR file** is a single file created to conglomerate any number of other files of any kind. A **JAR file** may hold **.java** files, **.class** files, **.jpg** files, **.wav** files, **.doc** files and/or anything else. If you think of a **JAR file** as a **zip** file, you have the right idea. When you create a **JAR file**, you specify whether or not you want compression. Compression is managed exactly as for **.zip** files; same algorithm and same format. In fact, the contents of a **.jar** file may be restored by the software that restores **.zip** files. Take a **.jar** file, rename it with the **.zip** extension, click on it, and your unzipping software unzips it.

Typically, you "**JAR-up**" all the files that an application or an applet will need in order to run (**.class** files, image files, and sound files) and distribute or transmit the single **.jar** file. The recipient may extract the original files and/or run the program directly from the **JAR file**, without extraction. One of the authors wanted to post on Blackboard around 15 Java source code files used by a software engineering application. Instead of uploading and labeling each of these files individually, it was much easier to put them into a **JAR file** and post that. Receiving the 15 files as a single **JAR file** was more convenient for the students as well. It spared them the effort of 15 individual downloads. In this essay as elsewhere, the terms **JAR file** and **.jar** file are synonymous.

Setting up Java applications so that can be launched by double-clicking an icon on the desktop is definitely the most captivating use of **JAR files** for beginning students. Being able to do this overcomes the disadvantage of being unable to create executable files (i.e. **.exe** files) from programs written in Java programs.

The program for creating a **JAR file** is named **jar**. It comes with the Java 2 Standard Edition just like the **javac**, **java**, **javadoc**, and **appletviewer** programs. For the purpose of demonstration, imagine we have the two files of Java source code shown below and their **.class** files saved in the folder **E:\JavaPrograms**. (To get the **.class** files, we needed to compile them; and so will you.)

```
//file: E:\JavaPrograms\Main.java

class Main
{
    public static void main(String[] args)
    {
        System.out.println("Calling LineMaker method from main()");
        LineMaker.lineOfAsterisks(60);
    }
}
```

```
//file: E:\JavaPrograms\LineMaker.java

class LineMaker
{
    static void lineOfAsterisks(int numberOfAsterisks)
    {
        System.out.println();
        for (int i = 0; i < numberOfAsterisks; i++)
        {
            System.out.print('*');
        }
        System.out.println();
    }
}
```
Action i: to create a JAR file named Test1.jar containing the files Main.class and LineMaker.class, compressed, use the following command:

```
E:\JavaProgs>jar cvf Test1.jar Main.class LineMaker.class
```

This shows the set-up for creating a JAR file using jar at the Command Prompt.

The arguments `cvf` specify that the action is to create a Java archive; to be verbose, meaning that a line on the console reports as each file is added to the JAR file; and that a file is to be written.

The name to be given to the created file follows the `cvf` arguments. JAR files typically have the `.jar` extension, but this is not a requirement.

Finally follows the list of files to be included in the created JAR. These may be spelled out, as done here, or designated using DOS's `*` (explained below).

The created file, Test1.jar, is written into the active directory.

* is the wildcard symbol, standing for "all." Thus, `*.class` stands for all `.class` files in the active directory. The command below includes all files in the created JAR file from the active directory that have the extension `.class` (and no others):

```
E:\JavaProgs>jar cvf Test12.jar *.class
```

The command below would include all `.java` files (and no others):

```
E:\JavaProgs>jar cvf Test13.jar *.java
```

The command below would include all `.java` files and all `.class` files:

```
E:\JavaProgs>jar cvf Test14.jar *.java *.class
```

The command below would include files with all names and all extension (that is, all files in the active directory):

```
E:\JavaProgs>jar cvf Test15.jar **
```
When you actually execute the command to put Main.class and LineMaker.class into Test1.jar, here is "the verbosity" you'll see. (If you wish to replicate this demonstration exactly, remember first to compile the given source code so that you will have the .class files.)

E:\JavaPrograms>jar cvf Test1.jar Main.class LineMaker.class
added manifest
adding: Main.class (in = 497) (out = 341) (deflated 31%)
adding: LineMaker.class (in = 441) (out = 317) (deflated 28%)

```
LineMaker.class was 441 bytes long.
Its compressed length is 317 bytes,
which is a 28% reduction in size.
```

E:\JavaPrograms>

The file Test1.jar was created and is now in the active directory, same folder as the .class files, which is E:\JavaPrograms. Test1.jar holds Main.class in compressed form and LineMaker.class in compressed form. It also holds a folder named META-INF (for meta information) which contains one file, MANIFEST.MF. MANIFEST.MF is a text file containing only the two lines shown below. You can ignore this entirely. It is not something you will need.

E:\JavaPrograms\META-INF>type manifest.mf
Manifest-Version: 1.0
Created-By: 1.5.0_04 (Sun Microsystems Inc.)

Action ii: to create a JAR file named Test2.jar containing the files Main.class and LineMaker.class, with no compression (that is, with "zero" compression), use the following command:

E:\JavaPrograms>jar c0vf Test2.jar Main.class LineMaker.class

Here is what you see when the command runs:

E:\JavaPrograms>jar c0vf Test2.jar Main.class LineMaker.class
added manifest
adding: Main.class (in = 497) (out = 497) (stored 0%)
adding: LineMaker.class (in = 441) (out = 441) (stored 0%)
E:\JavaPrograms>
Action iii: to create a JAR file named Test3.jar containing the files Main.class and LineMaker.class, compressed, with a manifest file that we have written and saved to specify that the application's starting point is the main() method in Main.class

This action, in other words, is to create a JAR file for an application in such a way that the application may be run directly from the JAR file. This JAR file may be compressed or not. Either way, it will run.

Before issuing the jar command to create the file, you need to write and save a "manifest file." This "manifest file" is a straight text file that may have any name and any extension. Suppose the file we write is named Manifest1.txt and that it is saved in the E:JavaPrograms folder along with the .class files to go into the JAR. All it needs is a single line to indicate the location of the Main-class, the name of the .class file holding the main() method with which the application is to begin. The following is perfect:

```
Main-class: Main
```

Main-class, with the hyphen, is a "keyword." It is not case sensitive, but the hyphen is essential. The only trap is the requirement that the final line in a manifest file, even a one-liner such as the one here, must be terminated with a linefeed (by striking enter).

If the last line in a manifest file is not terminated by striking enter, the jar facility will seem to have completed successfully, but the created file JAR file will not run. When you try, you will be informed of a failure to load the Main-Class manifest attribute.

Below is the command to create the runnable JAR file:

```
E:\JavaPrograms>jar cvmf Manifest1.txt Test3.jar Main.class LineMaker.class
```

The m and the f in the cvmf parameter list may be juxtaposed so that the parameter list reads cvfm. When the m comes before the f, as it does above, the name of the manifest file must come before the name of the .jar file to be created. When the f comes before the m, the name for the .jar must come before the name of the manifest file. The command with the juxtapositions would look as follows, and it performs identically to the one above:

```
E:\JavaPrograms>jar cvfm Test3.jar Manifest1.txt Main.class LineMaker.class
```

f and m are switched around names of these two files are switched around
Action iv: to run the application in the JAR file created by Action iii (that is, to run Test3.jar)

the same java command
that runs an application in
a .class file

command line argument
(note the dash)

the .jar file incorporating a manifest file
that specifies the class residence
of the starting main() method

E:\JavaPrograms>java -jar Test3.jar

A JAR file is runnable whether or not its contents were compressed.

Action v: to list the files in a JAR file (here we list the files in Test3.jar)

E:\JavaPrograms>jar tvf Test3.jar

Action vi: to extract the files in a JAR file (here, we'll extract the files in Test3.jar)

E:\JavaPrograms>jar xvf Test3.jar

You will notice that the manifest file is in the folder E:\JavaPrograms\META-INF and is named MANIFEST.MF. This is the name given to the enJarred manifest file regardless of the name you used when you created it.

Action vii: to make an application double-clickable (e.g. launchable from the desktop)

Action iii shows how to make a runnable .jar file (i.e. create a JAR file that includes a manifest file specifying the Main-class). A runnable application is double-clickable if it has a graphical user interface (that is, if runs in a window as opposed to the DOS console). Here are all the steps:

Create a .jar file, including everything the application will need to run as well as a manifest file specifying the Main-class. Locate this .jar file in MY COMPUTER, then right-click on it and select CREATE SHORTCUT. Drag the shortcut to the desktop. Right click on the shortcut; select OPEN WITH; and click JAVA (TM) 2 PLATFORM STANDARD EDITION BINARY.

Friends to whom you send this JAR file will need to have the JRE (the Java Runtime Environment) on their machine and specify (or have specified) that .jar files are to be opened with it (i.e. the JAVA (TM) 2 PLATFORM STANDARD EDITION BINARY.
Test this with the following program. Remember, it's the .class file that is needed for running, so you will need to compile the .java file. Also remember to create a manifest file specifying HelloGUI as the Main-class (and remember the linefeed).

```java
import javax.swing.JFrame;

public class HelloGUI
{
    public static void main(String[] args)
    {
        JFrame window = new JFrame();
        window.setTitle("HelloGUI");
        window.setSize(400, 200);
        window.setVisible(true);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

When the application and its resources (e.g. the .class files, .jpeg, etc) are stored in a package, the procedure for creating a runnable JAR file is only slightly more complicated. Suppose that everything needed to run is stored in package named p4, which is in the folder C: \ holder4. (This includes the compiled .class files). The source code with the main() method opens like this:

```java
// file: C: \ holder4\p4\HelloGUI4.java
package p4;
import javax.swing.JFrame;
public class HelloGUI4
{
    public static void main(String[] args)
    {
        // details omitted
    }
}
```

First, type-in the manifest file. It is still a one-liner specifying the Main-class. The difference is that name of the .class file holding the main() method is preceded by the name of its package. For the HelloGUI4 application, we'd type the manifest file in Notepad and it would look exactly as shown below. Remember to strike enter at the end of the line.

```
Main-class: p4.HelloGUI4
```

22
Second, save the manifest file; but not in the package's folder. Save it in the package's parent folder, in this case C:\holder4. Suppose it is called manifest.txt.

Third, in the Command Prompt window (DOS), make the active directory the package's parent folder, holder4. (A tutorial on changing directories is in an appendix at the end of this paper.)

active directory
is the package's parent folder

\c:\holder4>

Fourth and finally, enter the jar command to create the JAR file. The command is just like the one for action iii on page 20. The only difference is that the package's name appears on the command line, as shown below, instead of the list of files. The created JAR file may have any name, but the convention is to give it the same name as the package. It is placed into the active directory, which is the folder that holds the package. To put all files within the p4 package into the JAR file, the command is as follows:

active directory
is the package's parent folder

name of the JAR file
to be created

C:\holder4>jar cvmf manifest.txt p4.jar p4

p4.jar is double-clickable.

We have already warned you to watch out for a got-ya. A double-clickable JAR file is not an .exe file. To run, it requires the presence of the Java Runtime Environment (the JRE, also known as the bytecode interpreter, which is the java program called from the command line to run a compiled program). You have this because it comes with Java, the Java 2 Standard Edition Software Development Kit that you downloaded from Sun. Our guess is that the JRE is on most computers that have a browser. If a friend cannot run a tested, double-clickable application; either the computer lacks the JRE or has not been set to open a .jar file with the JAVA (TM) 2 PLATFORM STANDARD EDITION BINARY. The JRE can be downloaded separately from the full SDK from the Sun's Website.
A vendor may sell a special purpose library in the form of a package distributed as a **JAR** file. Uploading and downloading a single file is far less cumbersome than dealing with a welter of **.class** files individually. In addition, the **JAR** file is convenient to use inasmuch as the **.class** files within the en-JAR-ed package are accessible to the user without the need for extraction. The user's source code simply **imports** the ad hoc package in the usual way.

The extra step is that the **JAR** file needs to be on the user's **classpath**. Note carefully what we said. It is the **.jar** file itself that needs to be on the classpath in order for the user to both compile and to run the program; and when we say the **.jar** file we mean its complete path, starting with device.

The **JAR** file may be placed on the classpath in two ways. If the downloaded package is going to be used regularly, for the sake of convenience you should enter it onto the system's classpath via **CONTROL PANEL | SYSTEM PROPERTIES | ADVANCED | ENVIRONMENT VARIABLES**. This will allow you to import it in any of your files as easily as importing **java.util**. The other approach is to specify the classpath for the current run of **javac** and the current run of **java** on the command line with the **-cp** option. The catch is that an explicitly specified classpath overrides the classpath stored as the environment variable. This makes it necessary to add any needed locations, including the dot for the **.class** files in the active directory. As always, individual paths are separated from each other by a semicolon, and no spaces are allowed.

For a demonstration, imagine you've downloaded **howdy.jar**, created to hold all files in the vendor's package **howdy**. One of the files included is the following:

```java
package howdy;

import javax.swing.JFrame;

public class Hello
{
    public static void gui()
    {
        JFrame window = new JFrame();
        window.setTitle("Howdy Do");
        window.setSize(400, 200);
        window.setLocation(200, 75);
        window.setVisible(true);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

If you wish to test these instructions, remember to situate the **Hello.java** in a folder named **howdy** and compile it so that the **.class** file is in this folder as well. Create the **JAR** file so that it contains **Hello.class** and is named **howdy.jar**. No manifest file is needed.

You do **not** need to save the **JAR** file, **howdy.jar** (which we're pretending was downloaded), in a folder with the package's name. You may save it in any folder with any name you wish. Suppose you save it in **C:\vendor3\tools** so that its complete path is **C:\vendor3\tools\howdy.jar**.
Now, suppose you have the following program, which you have written, on your flash drive, which is device E: . Notice that it is importing the `Hello.class` file from the alleged vendor's `howdy` package.

```java
//file E:\Hey.java
import howdy.Hello;
public class Hey
{
    public static void main(String[] args)
    {
        System.out.println("Hey starting");
        Hello.gui();
        System.out.println("Hey ending");
    }
}
```

If you were to edit the stored classpath, appending `howdy.jar`, the stored classpath might well look like this (we're assuming all that had been on it before was the dot):

```
from before separator the JAR file that we added to the classpath
       ↓     ↓
       ↓     ↓
.\C:\vendor3\tools\howdy.jar
```

Then, your `Hey` program could be compiled and run as follows:

```bash
E:-->javac Hey.java
E:-->java Hey
```

If you were not to modify the system's classpath but, instead, to include the classpath on the command line, you would compile and run the program like this:

```bash
E:-->javac -cp C:\vendor3\tools\howdy.jar Hey.java
E:-->java -cp C:\vendor3\tools\howdy.jar; Hey
```

*The semicolon is the path separator (entry delimiter). The dot stands for the active directory, which is where `Hey.class` is found.*

Items along a classpath may be in any order.
The following applet, `PrettyHello.java`, is in the package `neat`. Suppose that it is compiled, so that `PrettyHello.class` is also in the `neat` folder, and the whole package is put into the compressed JAR file named `neat.jar`. No manifest file is needed, so the command to create the JAR file is:

```
E:\> jar cvf neat.jar neat
```

(recall, the JAR file is created from the directory that stores the package).

```java
// File: E:\neat\PrettyHello.java

package neat;

import javax.swing.*;
import java.awt.*;

public class PrettyHello extends JApplet
{
    Font font1 = new Font("TimesRoman", Font.BOLD, 50);
    Font font2 = new Font("Serif", Font.BOLD + Font.ITALIC, 50);
    Font font3 = new Font("SansSerif", Font.PLAIN, 50);

    public void paint(Graphics screen)
    {
        setBackground(Color.yellow); // background color
        screen.setColor(Color.red); // text's color

        screen.setFont(font1);
        screen.drawString("Hello World", 10, 60);

        screen.setFont(font2);
        screen.drawString("Hello World", 10, 100);

        screen.setFont(font3);
        screen.drawString("Hello World", 10, 140);
    }
}
```

Suppose that we received `neat.jar` via a download, and that we stored it in `C:\MyApplets` so that its full name is `C:\MyApplets\neat.jar`.

To run the applet directly from the JAR file, use the following HTML tags. Note that the .html file that launches the applet may be located on any device at the top level or in the folder of your choice:

```
<applet
    code = "neat.PrettyHello.class"
    archive = "C:\MyApplets\neat.jar"
    width = 325
    height = 160 >
</applet>
```

For the purpose of a test, copy the code (that is, the `applet` tags to the left) into Notepad and save the file as `E:\test.html`.

The applet can be launched with your browser or the SDK's `appletviewer`.

If these tags were in the file `E:\test.html`, the applet could be run with SDK's `appletviewer` as shown below:

```
E:\> appletviewer test.html
```
6) I/O between your computer and another computer

If both computers are on a network, they can talk. The mechanics for accomplishing this comes under the heading of client-server programming, socket programming, or network programming. Surprisingly, you need to know next to nothing about data communications. The most useful background would be acquaintance with reading Strings from the keyboard or reading and writing Strings to files because basic client-server programming is almost no different. There are just a couple of extra steps to set-up the connection.

You will need to write two coordinated programs, one for your computer and one for the other. The program that plays the part of the server is the one that "gets the call." It has to be running, awaiting the call from the computer which plays the part of the client. Waiting for "an input" from the client is the same kind of thing as a program's "blocking on input from the keyboard" (that is, waiting for the user to type-in something and strike enter). The client program, on the other computer, has to have the server program’s “telephone number,” which is the Internet Protocol address (IP address) of the computer on which it is running, and its "extension," which is the port on which you’ve coded the program to conduct its input and output. The server program does not need to have the client's IP address in the code, but it does need the port number. For the port number, any number between 2000 and 5000 should be fine. (We chose 2007 simply because that is the year.)

IP addresses are those "four, 8-bit, unsigned numbers expressed in base ten and written in dotted-string notation" such as 198.105.44.27. Let's say that the server program will be running on your computer. To get its IP address, so that you can code it into the client's program, use the DOS command ipconfig from any directory as shown below. That computer's IP address is 172.20.70.106. We'll use this in our code.

Our first pair of programs are written so that the code relating to communications is as uncluttered as possible. The server program starts and waits for the client. When the client program starts, it sends a String, entered as a literal on line 23 (page 29), to the server program. The server converts it to uppercase and sends it back to the client. Both programs then close.
Here is the server's program. An explanation of the follows.

```java
/* 1 */ import java.net.ServerSocket;
/* 2 */ import java.net.Socket;
/* 3 */ import java.io.InputStream;
/* 4 */ import java.io.InputStreamReader;
/* 5 */ import java.io.BufferedReader;
/* 6 */ import java.io.OutputStream;
/* 7 */ import java.io.PrintWriter;
/* 8 */
/* 9 */ class Server
/* 10 */ {
/* 11 */     public static void main(String[] args) throws Exception
/* 12 */     {
/* 13 */         int port = 2007; //server will await clients on port number 2007
/* 14 */         ServerSocket serverSocket = new ServerSocket(port);
/* 15 */         System.out.println("Server has opened on port " + port);
/* 16 */         Socket connectionToClient = serverSocket.accept();
/* 17 */         System.out.println("Client has made socket connection");
/* 18 */         InputStream inputStream = connectionToClient.getInputStream();
/* 19 */         InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
/* 20 */         BufferedReader inFromClient = new BufferedReader(inputStreamReader);
/* 21 */         OutputStream outputStream = connectionToClient.getOutputStream();
/* 22 */         PrintWriter outToClient = new PrintWriter(outputStream);
/* 23 */         String lineOfInputFromClient = inFromClient.readLine();
/* 24 */         System.out.println("Received from client: " + lineOfInputFromClient);
/* 25 */         String s = lineOfInputFromClient.toUpperCase(); //send back received string in upper case
/* 26 */         outToClient.println( s ); //make the transmission take place now
/* 27 */         System.out.println("Sent to client: " + s );
/* 28 */         if (connectionToClient != null) connectionToClient.close();
/* 29 */         if (inFromClient != null) inFromClient.close();
/* 30 */         if (outToClient != null) outToClient.close();
/* 31 */     }
/* 32 */ }
```

Line 14 creates a `ServerSocket` on port number 2007. The `ServerSocket`'s only use is on line 17 where, upon executing `serverSocket.accept()`, the program's flow stops until there is contact from a client. This "blocking on contact from a client" is similar to a `BufferedReader` object's blocking on `readLine()` until the user at the keyboard strikes return.

When a client arrives, `serverSocket.accept()` returns what may be thought of as a connecting linkage for communications between the server and the client. This linkage, which is actually a `Socket` object, is used on line 20 to get an `InputStream` from the client and on line 24 to get an `OutputStream` to the client. (In the server's program, input is from the client into the server. Output is from the server out to the client.)

The `InputStream` object gets wrapped in an `InputStreamReader` which gets wrapped in a `BufferedReader` (lines 21 – 22), whereupon capturing `Strings` from the client, with `readLine()` (line 27) is identical to capturing `Strings` from the keyboard.

The `OutputStream` object is wrapped in `PrintWriter` (line 25), whereupon writing `Strings` to the client, with `println()` and `println()` (line 31) is identical to writing `Strings` to a file.

Flushing (line 32) insures that the output is performed, even if the `PrintWriter`'s buffer is not full.
Here is the client's program:

```java
/* 1 */ import java.net.Socket;
/* 2 */ import java.io.InputStream;
/* 3 */ import java.io.InputStreamReader;
/* 4 */ import java.io.BufferedReader;
/* 5 */ import java.io.OutputStream;
/* 6 */ import java.io.PrintWriter;
/* 7 */
/* 8 */ class Client
/* 9 */ {
/* 10 */     public static void main(String[] args) throws Exception
/* 11 */     {
/* 12 */         int port = 2007;
/* 13 */         Socket connectionToServer = new Socket("172.20.70.106", port);
/* 14 */         // Socket connectionToServer = new Socket("localhost", port);
/* 15 */         InputStream inputStream = connectionToServer.getInputStream();
/* 16 */         InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
/* 17 */         BufferedReader inFromServer = new BufferedReader(inputStreamReader);
/* 18 */         /*
/* 19 */         OutputStream outputStream = connectionToServer.getOutputStream();
/* 20 */         PrintWriter outToServer = new PrintWriter(outputStream);
/* 21 */         String stringToSend = "Hello everybody, yes indeed!";
/* 22 */         */
/* 23 */         outToServer.println(stringToSend);
/* 24 */         outToServer.flush();
/* 25 */         System.out.println("sent to server: " + stringToSend);
/* 26 */         */
/* 27 */         String stringReceived = inFromServer.readLine();
/* 28 */         System.out.println("received from server: " + stringReceived);
/* 29 */         */
/* 30 */         if (connectionToServer != null) connectionToServer.close();
/* 31 */         if (inFromServer != null) inFromServer.close();
/* 32 */         if (outToServer != null) outToServer.close();
/* 33 */         } /*
/* 34 */     }
/* 35 */
/* 36 */
```

The process of creating a new `Socket`, on line 13, establishes the connection to the server. The server is on the computer which has the IP address of 172.20.70.106, and the server is "taking calls from clients" on port number 2007. (Ignore line 14 for now.)

The connection to the server is used just as the connection to the client was used in the previous program. It is used on line 16 to get an `InputStream` from the server and on line 20 to get an `OutputStream` to the server. (In the client's program, input is from the server into the client. Output is from the client out to the server.)

The `InputStream` object gets wrapped in an `InputStreamReader` which gets wrapped in a `BufferedReader` (lines 17 – 18), whereupon Strings from the server are captured with `readLine()` (line 29).

The `OutputStream` object is wrapped in `PrintWriter` (line 21), whereupon writing Strings to the client, with `println()` and `println()` (line 25).

Flushing (line 26) insures that the output is performed, even if the `PrintWriter`'s buffer is not full. (IO involves overhead and performance is better with an economy of scale. It's the same idea as when moving. You make one trip with a full truck rather than ten trips with an almost empty truck.)
Although less satisfying than running these programs on two computers, they can be tested on one computer. Run the server program in one Command Prompt window and the client program in a second Command Prompt window. This is where line 14 in the client program may come into play. If the server and the client are running in different DOS windows (or in different Eclipse windows) on the same machine, the client may either use the machine's IP address as on line 13 or may replace the IP address with "localhost" as on line 14. Line 14 (using "localhost") enables the client and server programs to communicate even if the computer is not online.

An obvious enhancement is for the client to get the String for transmission from the user at the keyboard and back and forth exchange of Strings into a while loop so that communication continues until the client sends an agreed upon "stop" as the sentinel. That the client and the server need to interact in accordance with expectations built into the dovetailing programs is what is meant by a protocol. Below is such a client, again communicating with the server at 172.20.70.106 on port 2007.

```java
import java.net.Socket;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.OutputStream;
import java.io.PrintWriter;

class EnhancedClient
{
    public static void main(String[] args) throws Exception
    {
        int port = 2007;
        Socket connectionToServer = new Socket("172.20.70.106", port);
        // Socket connectionToServer = new Socket("localhost", port);

        InputStream inputStream = connectionToServer.getInputStream();
        InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
        BufferedReader inFromServer = new BufferedReader(inputStreamReader);

        OutputStream outputStream = connectionToServer.getOutputStream();
        PrintWriter outToServer = new PrintWriter(outputStream);

        // for keyboard input
        InputStreamReader keyboardInputStream = new InputStreamReader(System.in);
        BufferedReader keyboardReader = new BufferedReader(keyboardInputStream);

        System.out.println(" >"); // User prompted to enter a line
        String s = keyboardReader.readLine();

        while(!s.equalsIgnoreCase("stop"))
        {
            outToServer.println(s);
            outToServer.flush();

            s = inFromServer.readLine();
            System.out.println("From server: " + s);

            System.out.println(" >");
            s = keyboardReader.readLine();
        }

        outToServer.println(" "); // sends out the "stop"
        outToServer.flush();

        if (inFromServer != null) inFromServer.close();
        if (outToServer != null) outToServer.close();
        if (connectionToServer != null) connectionToServer.close();

        System.out.println("\nClient closing");
    }
}
```
Enhancing the server so that it can handle multiple clients at the same time involves putting communications with each client into a separate thread. A thread is an object that encapsulates an independent copy of a process. Just as every instance of the `StringTokenizer` class is a fully functional, autonomous `StringTokenizer`; every instance of the `Thread` class is an operational process. "Instance of the `Thread` class" includes instances of classes that are descendents from `Thread` inasmuch as an object from a class that extends `Thread` IS-A `Thread`.

It is only the server's code that needs threads, not the client's, because from the client's perspective only a single process is taking place -- its own communication with the server. A server that can manage a logically unlimited number of clients (coded to match the protocol of the `EnhancedClient` on the previous page) is shown below. It shows how a server with threads is architected. Because there are no critical section issues, the code is straightforward. Note that the program, in one file, `EnhancedServerOne.java`, is in two classes on this page and the next, `EnhancedServerOne` and `ServerProcessOne`.

```java
//File: EnhancedServerOne.java

//Working with the client on the previous page,
//this server is at IP address 172.20.70.106

import java.net.ServerSocket;
import java.net.Socket;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.io.PrintStream;

class EnhancedServerOne
{
    public static void main(String[] args) throws Exception
    {
        int port = 2007;
        ServerSocket serverSocket = new ServerSocket(port);
        System.out.println("Server has opened on port " + port);

        int clientNumber = 0;

        while(true)
        {
            Socket connectionToClient = serverSocket.accept();
            clientNumber = clientNumber + 1;
            System.out.println("Client number " + clientNumber +
            " has made socket connection\n");

            Thread thread = new ServerProcessOne(connectionToClient, clientNumber);
            thread.start();
        }
    }
}
```
class ServerProcessOne extends Thread
{
    Socket connectionToClient;
    int clientNumber;

    ServerProcess(Socket socket, int clientNumber) //constructor
    {
        connectionToClient = socket;
        this.clientNumber = clientNumber;
    }

    public void run()
    {
        try
        {
            InputStream inputStream = connectionToClient.getInputStream();
            InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
            BufferedReader inFromClient = new BufferedReader(inputStreamReader);

            OutputStream outputStream = connectionToClient.getOutputStream();
            PrintWriter outToClient = new PrintWriter(outputStream);

            String lineOfInputFromClient = inFromClient.readLine();

            while( ! lineOfInputFromClient.equalsIgnoreCase("stop") )
            {
                System.out.println(" From client#" + clientNumber + ": " + lineOfInputFromClient);
                lineOfInputFromClient = lineOfInputFromClient.toUpperCase();
                outToClient.println( lineOfInputFromClient );
                outToClient.flush();
                System.out.println("Sent to client#" + clientNumber + ": " + lineOfInputFromClient);
            }
            lineOfInputFromClient = inFromClient.readLine();

            System.out.println("Received from client#" + clientNumber + ": " + lineOfInputFromClient);
            if (inFromClient != null) inFromClient.close();
            if (outToClient != null) outToClient.close();
            if (connectionToClient != null) connectionToClient.close();

            System.out.println("Connection to client#" + clientNumber + " is closed.");
        }
        catch(Exception e)
        { }
    }
}
The enhanced server, above, may be rewritten so that the `ServerProcessOne` class can be replaced with a class that does not extend `Thread` but, instead, implements the `Runnable` interface. The `EnhancedServerTwo` program, below, illustrates. This alternative opens the option for the server's class to be descendent from an application-pertinent superclass.

`EnhancedServerTwo` works with the same `EnhancedClient` program from page 30 as `EnhancedServerOne` and it performs identically. The `Runnable` interface includes only one method, `run()`. It has the same signature as the `Thread`'s `run()` method: `public void run()`. It holds exactly the same code held by the `run()` method in the previous program. The differences in the code between this program and `EnhancedServerOne` are shown in bold. The replaced statements are commented out.

```java
//File: EnhancedServerTwo.java

//Working with the client on page 30, this server is at IP address 172.20.70.106

import java.net.ServerSocket;
import java.net.Socket;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.OutputStream;
import java.io.PrintWriter;

class EnhancedServerTwo {
    public static void main(String[] args) throws Exception {
        int port = 2007;
        ServerSocket serverSocket = new ServerSocket(port);
        System.out.println("Server has opened on port " + port);
        int clientNumber = 0;
        while(true) {
            Socket connectionToClient = serverSocket.accept();
            clientNumber = clientNumber + 1;
            System.out.println("Client number " + clientNumber + " has made socket connection\n");

            //Thread thread = new ServerProcessOne(connectionToClient, clientNumber);
            //thread.start();

            ServerProcessTwo serverProcessTwo =
                new ServerProcessTwo(connectionToClient, clientNumber);
            Thread thread = new Thread(serverProcessTwo);
            thread.start();
        }
    }
}
```
//class ServerProcessOne extends Thread
class ServerProcessTwo implements Runnable
{
    Socket connectionToClient;
    int clientNumber;

    ServerProcess(Socket socket, int clientNumber) //constructor
    {
        connectionToClient = socket;
        this.clientNumber = clientNumber;
    }

    public void run()
    {
        try
        {
            InputStream inputStream = connectionToClient.getInputStream();
            InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
            BufferedReader inFromClient = new BufferedReader(inputStreamReader);

            OutputStream outputStream = connectionToClient.getOutputStream();
            PrintWriter outToClient = new PrintWriter(outputStream);

            String lineOfInputFromClient = inFromClient.readLine();

            while( ! lineOfInputFromClient.equalsIgnoreCase("stop") )
            {
                System.out.println(" From client#" + clientNumber + ": " + lineOfInputFromClient);
                lineOfInputFromClient = lineOfInputFromClient.toUpperCase();
                outToClient.println( lineOfInputFromClient );
                outToClient.flush();
                System.out.println("Sent to client#" + clientNumber + ": " + lineOfInputFromClient);
                lineOfInputFromClient = inFromClient.readLine();
            }

            System.out.println("Received from client#" + clientNumber + ": " + lineOfInputFromClient);

            if (inFromClient != null) inFromClient.close();
            if (outToClient != null) outToClient.close();
            if (connectionToClient != null) connectionToClient.close();

            System.out.println("Connection to client#" + clientNumber + " is closed.");
        }
        catch(Exception e)
        {
        }
    }
}
7) Writing an applet so that it opens both as an applet and as an application

Any class may have a main() method; and its main() method is activated when the class is launched as an application (otherwise its main() method is ignored). To build an applet so that it also opens as an application, include a main() method that:

** creates a JFrame

** creates an instance of the ad hoc class (the JApplet or the Applet); we'll call it theApplet

** add()s theApplet to the JFrame's content pane

** lauches theApplet by calling init() on it and then calling start() on it

```java
import javax.swing.*;
import java.awt.*;

public class Applet_or_Application extends JApplet
{
    //the main method gets an instance of the applet and starts it

    public static void main(String[] args)
    {
        JFrame window = new JFrame("Program that Runs as an Application or an Applet");
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JApplet theApplet = new Applet_or_Application();

        Container contentPane = window.getContentPane();
        contentPane.add( theApplet );

        theApplet.init();
        theApplet.start();

        window.setSize(750, 250);
        window.setVisible(true);
    }

    public void paint(Graphics screen)
    {
        Font font1 = new Font("TimesRoman", Font.BOLD, 50);
        Font font2 = new Font("Serif", Font.BOLD + Font.ITALIC, 50);
        Font font3 = new Font("SansSerif", Font.PLAIN, 50);

        setBackground(Color.yellow);  //background color
        screen.setColor(Color.red);   //text's color

        screen.setFont(font1);
        screen.drawString("Opens as Applet or Application", 10, 60);

        screen.setFont(font2);
        screen.drawString("Opens as Applet or Application", 10, 110);

        screen.setFont(font3);
        screen.drawString("Opens as Applet or Application", 10, 160);
    }
}
```
This program may be compiled and run just as any application may be. To run as an applet, as always, an .html (or .htm) file is needed with an opening and closing applet tag, such as the following:

```xml
<applet
    code   = "Applet_or_Application.class"
    width  = 750
    height = 250
</applet>
```
Appendix: Notes on the CD (Change Directory) and MD (Make Directory) commands

The Command Prompt application (DOS) is located under

Start | Programs | Accessories

When the Command Prompt application opens, the prompt will look something like this:

```
The DOS prompt

C:\Documents and Settings\Owner>
```

The terms folder and directory mean the same thing. The active folder is the location where javac, java, javadoc, jar, and the appletviewer tend to look for files as the default.

The active folder (i.e. the active directory) is changed with the CD (Change Directory) command. CD\ makes active the device's root directory. Commands are not case sensitive, so CD cd Cd are equivalent.

```
C:\Documents and Settings\Owner> cd\
gives C:\>
```

To make C:\holder1\pl the active directory, that folder must exist. Suppose that it does. To change to it from any directory as the active directory on the same device, enter its complete path:

```
gives C:\Documents and Settings\Owner> cd c:\holder1\pl
```
or

```
gives C:\Documents and Settings\Owner> cd holder1\pl
```

37
With an ancestor directory as the active directory, one need only enter the path from the ancestor to the folder. 
C:\ is C:\holder's ancestor hence C:\holder\p1's ancestor. C:\holder is C:\holder\p1's ancestor. Therefore:

\texttt{C:\> cd holder1}
\texttt{\rightarrow C:\holder1>}

\texttt{C:\> cd holder1\p1}
\texttt{\rightarrow C:\holder1\p1>}

\texttt{C:\holder1> cd p1}
\texttt{\rightarrow C:\holder1\p1>}

To change to a different device, with any active directory, enter the device's letter followed by a colon with no space. (Many people consider the colon as part of the device's name.)

\texttt{C:\holder1\p1> e:}
\texttt{\rightarrow E:\>}

This switch does not change the active directory on C:, it is still C:\holder1\p1. Referring to the device as C: refers implicitly to its active directory. Below, the first two commands switch to device E: and make E:\pack1 the active directory on E:. The third command copies all .java files from the active directory, E:\pack1, to the active directory on C:, which is C:\holder1\p1.

\texttt{C:\holder1\p1> e:}
\texttt{E:\> cd pack1}
\texttt{E:\pack1> copy *.java c:}

The MD command makes a directory. You need never use it because making a folder through \textit{MY COMPUTER | FILE | NEW | FOLDER} accomplishes the same end.

MD places the new folder within the active directory. The following command creates the folder p2 within C:\holder1; in other words, it creates C:\holder1\p2 (but it does not change to that directory; for that you need a separate command).

\texttt{C:\holder1> md p2}

The command below creates three folders: C:\aa, C:\aa\bb, and C:\aa\bb\cc

\texttt{C:\>md aa\bb\cc}
The Ivan G. Seidenberg
School of Computer Science and Information Systems
Pace University
Technical Report Series

EDITORIAL BOARD

Editor:
Allen Stix, Computer Science, Pace--Westchester

Associate Editors:
Constance A. Knapp, Information Systems, Pace--Westchester
Susan M. Merritt, Dean, The Ivan G. Seidenberg School of Computer Science and Information Systems --Pace

Members:
Howard S. Blum, Computer Science, Pace--New York
Mary F. Courtney, Computer Science, Pace--Westchester
Nicholas J. De Lillo, Mathematics and Computer Science, Manhattan College
Daniel Farkas, Information Systems, Pace--Westchester
Fred Grossman, Information Systems; Doctor of Professional Studies, Pace--New York and White Plains
Fran Goertzel Gustavson, Information Systems, Pace--Westchester
Joseph F. Malerba, Computer Science, Pace--Westchester
John S. Mallozzi, Computer Information Sciences, Iona College
John C. Molluzzo, Information Systems, Pace--New York
Pauline Mosley, Technology Systems, Pace--New York
Narayan S. Murthy, Computer Science, Pace--New York
Catherine Ricardo, Computer Information Sciences, Iona College
Judith E. Sullivan, CSIS Alumna, MS in CS from Pace--Westchester
Sylvester Tuohy, Computer Science, Pace--Westchester

The Ivan G. Seidenberg School of Computer Science and Information Systems, through the Technical Report Series, provides members of the community an opportunity to disseminate the results of their research by publishing monographs, working papers, and tutorials. Technical Reports is a place where scholarly striving is respected.

All preprints and recent reprints are requested and accepted. New manuscripts are read by two members of the editorial board; the editor decides upon publication. Authors, please note that production is Xerographic from your submission. Statements of policy and mission may be found in issues #29 (April 1990) and #34 (September 1990).

Please direct submissions as well as requests for single copies to:

Allen Stix
The Ivan G. Seidenberg School of Computer Science and Information Systems
Goldstein Academic Center
Pace University
861 Bedford Road
Pleasantville, NY 10570-2799