UNIT III

Overview

The Java programming language and environment is designed to solve a number of problems in modern programming practice. Java started as a part of a larger project to develop advanced software for consumer electronics. These devices are small, reliable, portable, distributed, real-time embedded systems. When we started the project we intended to use C++, but encountered a number of problems. Initially these were just compiler technology problems, but as time passed more problems emerged that were best solved by changing the language.

Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.

Objectives::

- Describe how java has become more popular on internet.
- Explain JVM functions.
- Explain why applet is used in internet.
- Describe how server side programmings are used.

Java Programming

Java is a programming language originally developed by James Gosling at Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture.

The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1995. As of May 2007, in compliance with the

specifications of the Java Community Process, Sun made available most of their Java technologies as free software under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java and GNU Classpath.

History

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later renamed as Java, from a list of random words. Gosling aimed to implement a virtual machine and a language that had a familiar C/C++ style of notation.

Sun released the first public implementation as Java 1.0 in 1995. It promised "Write Once, Run anywhere" (WORA), providing no-cost run-times on popular platforms. Fairly secure and featuring configurable security, it allowed network- and file-access restrictions. Major web browsers soon incorporated the ability to run Java applets within web pages, and Java quickly became popular. With the advent of Java 2 (released initially as J2SE 1.2 in December 1998), new versions had multiple configurations built for different types of platforms. For example, J2EE targeted enterprise applications and the greatly stripped-down version J2ME for mobile applications. J2SE designated the Standard Edition. In 2006, for marketing purposes, Sun renamed new J2 versions as Java EE, Java ME, and Java SE, respectively.

In 1997, Sun Microsystems approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a de facto standard, controlled through the Java Community Process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) (a subset of the SDK); the primary distinction involves the JRE's lack of the compiler, utility programs, and header files.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL). On 8 May 2007 Sun finished the process, making all of Java's core code available under free software / open-source distribution terms, aside from a small portion of code to which Sun did not hold the copyright.

Philosophy

Primary goals

There were five primary goals in the creation of the Java language:

1. It should be "simple, object oriented, and familiar".
2. It should be "robust and secure".
3. It should be "architecture neutral and portable".
4. It should execute with "high performance".
5. It should be "interpreted, threaded, and dynamic".

Java Platform

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any supported hardware/operating-system platform. One should be able to write a program once, compile it once, and run it anywhere.

This is achieved by compiling the Java language code, not to machine code but to Java bytecode – instructions analogous to machine code but intended to be interpreted by a virtual machine (VM) written specifically for the host hardware. End-users commonly use a Java Runtime Environment (JRE) installed on their own machine for standalone Java applications, or in a Web browser for Java applets.

Standardized libraries provide a generic way to access host specific features such as graphics, threading and networking. In some JVM versions, bytecode can be compiled to native code, either before or during program execution, resulting in faster execution.

A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would, and Java suffered a reputation for poor performance. This gap has been narrowed by a number of optimization techniques introduced in the more recent JVM implementations.

One such technique, known as just-in-time (JIT) compilation, translates Java bytecode into native code the first time that code is executed, then caches it. This results in a program that starts and executes faster than pure interpreted code can, at the cost of introducing occasional compilation overhead during execution. More sophisticated VMs also use dynamic recompilation, in which the VM analyzes the behavior of the running program and selectively recompiles and optimizes parts of the program. Dynamic recompilation can achieve optimizations superior to static compilation because the dynamic compiler can base optimizations on knowledge about the runtime environment and the set of loaded classes, and can identify hot spots - parts of the program, often inner loops, that take up the most execution time. JIT compilation and dynamic recompilation allow Java programs to approach the speed of native code without losing portability.

Another technique, commonly known as static compilation, or ahead-of-time (AOT) compilation, is to compile directly into native code like a more traditional compiler. Static Java compilers translate the Java source or bytecode to native object code. This achieves good performance compared to interpretation, at the expense of portability; the output of these compilers can only be run on a single architecture. AOT could give Java

something close to native performance, yet it is still not portable since there are no compiler directives, and all the pointers are indirect with no way to micro manage garbage collection.

Java's performance has improved substantially since the early versions, and performance of JIT compilers relative to native compilers has in some tests been shown to be quite similar. The performance of the compilers does not necessarily indicate the performance of the compiled code; only careful testing can reveal the true performance issues in any system.

One of the unique advantages of the concept of a runtime engine is that even the most serious errors (exceptions) in a Java program should not 'crash' the system under any circumstances, provided the JVM itself is properly implemented. Moreover, in runtime engine environments such as Java there exist tools that attach to the runtime engine and every time that an exception of interest occurs they record debugging information that existed in memory at the time the exception was thrown (stack and heap values). These Automated Exception Handling tools provide 'root-cause' information for exceptions in Java programs that run in production, testing or development environments. Such precise debugging is much more difficult to implement without the run-time support that the JVM offers.

Implementations

Sun Microsystems officially licenses the Java Standard Edition platform for Microsoft Windows, Linux, Mac OS X, and Solaris. Through a network of third-party vendors and licensees, alternative Java environments are available for these and other platforms.

Sun's trademark license for usage of the Java brand insists that all implementations be "compatible". This resulted in a legal dispute with Microsoft after Sun claimed that the Microsoft implementation did not support RMI or JNI and had added platform-specific features of their own. Sun sued in 1997, and in 2001 won a settlement of $20 million as well as a court order enforcing the terms of the license from Sun. As a result, Microsoft no longer ships Java with Windows, and in recent versions of Windows, Internet Explorer cannot support Java applets without a third-party plugin. Sun, and others, have made available free Java run-time systems for those and other versions of Windows.

Platform-independent Java is essential to the Java EE strategy, and an even more rigorous validation is required to certify an implementation. This environment enables portable server-side applications, such as Web services, servlets, and Enterprise JavaBeans, as well as with embedded systems based on OSGi, using Embedded Java environments. Through the new GlassFish project, Sun is working to create a fully functional, unified open-source implementation of the Java EE technologies.

Sun also distributes a superset of the JRE called the Java Development Kit (commonly known as the JDK), which includes development tools such as the Java compiler, Javadoc, Jar and debugger.

Automatic memory management

Java uses an automatic garbage collector to manage memory in the object lifecycle. The programmer determines when objects are created, and the Java runtime is responsible for recovering the memory once objects are no longer in use. Once no references to an object remain, the unreachable object becomes eligible to be freed automatically by the garbage collector. Something similar to a memory leak may still occur if a programmer's code holds a reference to an object that is no longer needed, typically when objects that are no longer needed are stored in containers that are still in use. If methods for a nonexistent object are called, a "null pointer exception" is thrown.

One of the ideas behind Java's automatic memory management model is that programmers be spared the burden of having to perform manual memory management. In some languages memory for the creation of objects is implicitly allocated on the stack, or explicitly allocated and deallocated from the heap. Either way the responsibility of managing memory resides with the programmer. If the program does not deallocate an object, a memory leak occurs. If the program attempts to access or deallocate memory that has already been deallocated, the result is undefined and difficult to predict, and the program is likely to become unstable and/or crash. This can be partially remedied by the use of smart pointers, but these add overhead and complexity.

Garbage collection may happen at any time. Ideally, it will occur when a program is idle. It is guaranteed to be triggered if there is insufficient free memory on the heap to allocate a new object; this can cause a program to stall momentarily. Where performance or response time is important, explicit memory management and object pools are often used.

Java does not support C/C++ style pointer arithmetic, where object addresses and unsigned integers (usually long integers) can be used interchangeably. This allows the garbage collector to relocate referenced objects, and ensures type safety and security.

As in C++ and some other object-oriented languages, variables of Java's primitive types are not objects. Values of primitive types are either stored directly in fields (for objects) or on the stack (for methods) rather than on the heap, as commonly true for objects (but see Escape analysis). This was a conscious decision by Java's designers for performance reasons. Because of this, Java was not considered to be a pure object-oriented programming language. However, as of Java 5.0, autoboxing enables programmers to proceed as if primitive types are instances of their wrapper classes.

Syntax

The syntax of Java is largely derived from C++. Unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built almost exclusively as an object oriented language. All code is written inside a class and everything is an object, with the exception of the intrinsic data types (ordinal and real numbers, boolean values, and characters), which are not classes for performance reasons.

Java suppresses several features (such as operator overloading and multiple inheritance) for classes in order to simplify the language and to prevent possible errors and anti-pattern design.

Java uses the same commenting methods as C++. There are two different styles of comment: a single line style marked with two forward slashes, and a multiple line style opened with a forward slash asterisk (/*) and closed with an asterisk forward slash (*/).

Example:

```
//This is an example of a single line comment using two forward slashes

/* This is an example of a multiple line comment using the forward slash
   and asterisk. This type of comment can be used to hold a lot of information
   but it is very important to remember to close the comment. */
```

Examples

Hello world

The traditional Hello world program can be written in Java as:

```
/*
 * Outputs "Hello, world!" and then exits
 */

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

By convention, source files are named after the public class they contain, appending the suffix .java, for example, HelloWorld.java. It must first be compiled into bytecode, using a Java compiler, producing a file named HelloWorld.class. Only then can it be executed, or 'launched'. The java source file may only contain one public class but can contain multiple classes with less than public access and any number of public inner classes.

A **class** that is declared **private** may be stored in any .java file. The compiler will generate a class file for each class defined in the source file. The name of the class file is the name of the class, with .class appended. For class file generation, anonymous classes are treated as if their name was the concatenation of the name of their enclosing class, a $, and an integer.

The keyword **public** denotes that a method can be called from code in other classes, or that a class may be used by classes outside the class hierarchy. The class hierarchy is related to the name of the directory in which the .java file is.

The keyword **static** in front of a method indicates a static method, which is associated only with the class and not with any specific instance of that class. Only static methods can be invoked without a reference to an object. Static methods cannot access any method variables that are not static.

The keyword **void** indicates that the main method does not return any value to the caller. If a Java program is to exit with an error code, it must call System.exit() explicitly.

The method name "main" is not a keyword in the Java language. It is simply the name of the method the Java launcher calls to pass control to the program. Java classes that run in managed environments such as applets and Enterprise Java Beans do not use or need a main() method. A java program may contain multiple classes that have main methods, which means that the VM needs to be explicitly told which class to launch from.

The main method must accept an array of **String** objects. By convention, it is referenced as **args** although any other legal identifier name can be used. Since Java 5, the main method can also use variable arguments, in the form of public static void main(String... args), allowing the main method to be invoked with an arbitrary number of String arguments. The effect of this alternate declaration is semantically identical (the args parameter is still an array of String objects), but allows an alternate syntax for creating and passing the array.

The Java launcher launches Java by loading a given class (specified on the command line or as an attribute in a JAR) and starting its public static void main(String[]) method. Stand-alone programs must declare this method explicitly. The String[] args parameter is an array of String objects containing any arguments passed to the class. The parameters to main are often passed by means of a command line.

Printing is part of a Java standard library: The **System** class defines a public static field called **out**. The out object is an instance of the PrintStream class and provides many methods for printing data to standard out, including **println(String)** which also appends a new line to the passed string.

The string "Hello, world!" is automatically converted to a String object by the compiler.

A more comprehensive example

```
// OddEven.java
import javax.swing.JOptionPane;

public class OddEven {
   // "input" is the number that the user gives to the computer
   private int input; // a whole number("int" means integer)

   /*
    * This is the constructor method. It gets called when an object of the OddEven type
    * is being created.
```

```
 */
public OddEven() {
//Code not shown
}

// This is the main method. It gets called when this class is run through a Java interpreter.
public static void main(String[] args) {
   /*
    * This line of code creates a new instance of this class called "number" (also known as an
    * Object) and initializes it by calling the constructor.  The next line of code calls
    * the "showDialog()" method, which brings up a prompt to ask you for a number
    */
   OddEven number = new OddEven();
   number.showDialog();
}

public void showDialog() {
   /*
    * "try" makes sure nothing goes wrong. If something does,
    * the interpreter skips to "catch" to see what it should do.
    */
   try {
      /*
       * The code below brings up a JOptionPane, which is a dialog box
       * The String returned by the "showInputDialog()" method is converted into
       * an integer, making the program treat it as a number instead of a word.
       * After that, this method calls a second method, calculate() that will
       * display either "Even" or "Odd."
       */
      input = new Integer(JOptionPane.showInputDialog("Please Enter A Number"));
      calculate();
   } catch (NumberFormatException e) {
      /*
       * Getting in the catch block means that there was a problem with the format of
       * the number. Probably some letters were typed in instead of a number.
       */
      System.err.println("ERROR: Invalid input. Please type in a numerical value.");
   }
}

/*
 * When this gets called, it sends a message to the interpreter.
 * The interpreter usually shows it on the command prompt (For Windows users)
 * or the terminal (For Linux users).(Assuming it's open)
```

```
     */
    private void calculate() {
        if (input % 2 == 0) {
            System.out.println("Even");
        } else {
            System.out.println("Odd");
        }
    }
}
```

- The **import** statement imports the **JOptionPane** class from the **javax.swing** package.
- The **OddEven** class declares a single **private** field of type **int** named **input**. Every instance of the OddEven class has its own copy of the input field. The private declaration means that no other class can access (read or write) the input field.
- **OddEven()** is a **public** constructor. Constructors have the same name as the enclosing class they are declared in, and unlike a method, have no return type. A constructor is used to initialize an object that is a newly created instance of the class.
- The **calculate()** method is declared without the static keyword. This means that the method is invoked using a specific instance of the OddEven class. (The reference used to invoke the method is passed as an undeclared parameter of type OddEven named **this**.) The method tests the expression input % 2 == 0 using the **if** keyword to see if the remainder of dividing the input field belonging to the instance of the class by two is zero. If this expression is true, then it prints **Even**; if this expression is false it prints **Odd**. (The input field can be equivalently accessed as this.input, which explicitly uses the undeclared this parameter.)
- **OddEven number = new OddEven();** declares a local object reference variable in the main method named number. This variable can hold a reference to an object of type OddEven. The declaration initializes number by first creating an instance of the OddEven class, using the **new** keyword and the OddEven() constructor, and then assigning this instance to the variable.
- The statement **number.showDialog();** calls the calculate method. The instance of OddEven object referenced by the number local variable is used to invoke the method and passed as the undeclared this parameter to the calculate method.
- **input = new Integer(JOptionPane.showInputDialog("Please Enter A Number"));** is a statement that converts the type of **String** to the primitive type **int** by taking advantage of the wrapper class **Integer**.

Special classes

Applet

Java applets are programs that are embedded in other applications, typically in a Web page displayed in a Web browser.

```
// Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

The **import** statements direct the Java compiler to include the **javax.swing.JApplet** and **java.awt.Graphics** classes in the compilation. The import statement allows these classes to be referenced in the source code using the simple class name (i.e. JApplet) instead of the fully qualified class name (i.e. javax.swing.JApplet).

The Hello class **extends** (subclasses) the **JApplet** (Java Applet) class; the JApplet class provides the framework for the host application to display and control the lifecycle of the applet. The JApplet class is a JComponent (Java Graphical Component) which provides the applet with the capability to display a graphical user interface (GUI) and respond to user events.

The Hello class overrides the **paintComponent(Graphics)** method inherited from the Container superclass to provide the code to display the applet. The paint() method is passed a **Graphics** object that contains the graphic context used to display the applet. The paintComponent() method calls the graphic context **drawString(String, int, int)** method to display the **"Hello, world!"** string at a pixel offset of (**65, 95**) from the upper-left corner in the applet's display.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<!-- Hello.html -->
<html>
  <head>
    <title>Hello World Applet</title>
  </head>
  <body>
    <applet code="Hello" width="200" height="200">
    </applet>
  </body>
</html>
```

An applet is placed in an HTML document using the **<applet>** HTML element. The applet tag has three attributes set: **code="Hello"** specifies the name of the JApplet class and **width="200" height="200"** sets the pixel width and height of the applet. Applets may also be embedded in HTML using either the object or embed element, although

support for these elements by Web browsers is inconsistent. However, the applet tag is deprecated, so the object tag is preferred where supported.

The host application, typically a Web browser, instantiates the **Hello** applet and creates an AppletContext for the applet. Once the applet has initialized itself, it is added to the AWT display hierarchy. The paint method is called by the AWT event dispatching thread whenever the display needs the applet to draw itself.

Servlet

Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems. Servlets are server-side Java EE components that generate responses (typically HTML pages) to requests (typically HTTP requests) from clients. A servlet can almost be thought of as an applet that runs on the server side—without a face.

```
// Hello.java
import java.io.*;
import javax.servlet.*;

public class Hello extends GenericServlet {
   public void service(ServletRequest request, ServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      final PrintWriter pw = response.getWriter();
      pw.println("Hello, world!");
      pw.close();
   }
}
```

The **import** statements direct the Java compiler to include all of the public classes and interfaces from the **java.io** and **javax.servlet** packages in the compilation.

The **Hello** class **extends** the **GenericServlet** class; the GenericServlet class provides the interface for the server to forward requests to the servlet and control the servlet's lifecycle.

The Hello class overrides the **service(ServletRequest, ServletResponse)** method defined by the Servlet interface to provide the code for the service request handler. The service() method is passed a **ServletRequest** object that contains the request from the client and a **ServletResponse** object used to create the response returned to the client. The service() method declares that it **throws** the exceptions ServletException and IOException if a problem prevents it from responding to the request.

The **setContentType(String)** method in the response object is called to set the MIME content type of the returned data to **"text/html"**. The **getWriter()** method in the response

returns a **PrintWriter** object that is used to write the data that is sent to the client. The **println(String)** method is called to write the **"Hello, world!"** string to the response and then the **close()** method is called to close the print writer, which causes the data that has been written to the stream to be returned to the client.

JavaServer Page

JavaServer Pages (JSPs) are server-side Java EE components that generate responses, typically HTML pages, to HTTP requests from clients. JSPs embed Java code in an HTML page by using the special delimiters <% and %>. A JSP is compiled to a Java servlet, a Java application in its own right, the first time it is accessed. After that, the generated servlet creates the response.

Swing application

Swing is a graphical user interface library for the Java SE platform. It is possible to specify a different look and feel through the pluggable look and feel system of Swing. Clones of Windows, GTK and Motif are supplied by Sun. Apple also provides an Aqua look and feel for Mac OS X. Where prior implementations of these looks and feels may have been considered lacking, Swing in Java SE 6 addresses this problem by using more native widget drawing routines of the underlying platforms.

This example Swing application creates a single window with "Hello, world!" inside:

```
// Hello.java (Java SE 5)
import java.awt.BorderLayout;
import javax.swing.*;

public class Hello extends JFrame {
   public Hello() {
      super("hello");
      setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
      setLayout(new BorderLayout());
      add(new JLabel("Hello, world!"));
      pack();
   }

   public static void main(String[] args) {
      new Hello().setVisible(true);
   }
}
```

The first **import** statement directs the Java compiler to include the BorderLayout class from the java.awt package in the compilation; the second **import** includes all of the public classes and interfaces from the **javax.swing** package.

The **Hello** class **extends** the **JFrame** class; the JFrame class implements a window with a title bar and a close control.

The **Hello()** constructor initializes the frame by first calling the superclass constructor, passing the parameter "hello", which is used as the window's title. It then calls the **setDefaultCloseOperation(int)** method inherited from JFrame to set the default operation when the close control on the title bar is selected to **WindowConstants.EXIT_ON_CLOSE** — this causes the JFrame to be disposed of when the frame is closed (as opposed to merely hidden), which allows the JVM to exit and the program to terminate. Next, the layout of the frame is set to a BorderLayout; this tells Swing how to arrange the components that will be added to the frame. A **JLabel** is created for the string **"Hello, world!"** and the **add(Component)** method inherited from the Container superclass is called to add the label to the frame. The **pack()** method inherited from the Window superclass is called to size the window and lay out its contents, in the manner indicated by the BorderLayout.

The **main()** method is called by the JVM when the program starts. It instantiates a new **Hello** frame and causes it to be displayed by calling the **setVisible(boolean)** method inherited from the Component superclass with the boolean parameter **true**. Once the frame is displayed, exiting the main method does not cause the program to terminate because the AWT event dispatching thread remains active until all of the Swing top-level windows have been disposed.

Generics

In 2004 generics were added to the Java language, as part of J2SE 5.0. Prior to the introduction of generics, each variable declaration had to be of a specific type. For container classes, for example, this is a problem because there is no easy way to create a container that accepts only specific types of objects. Either the container operates on all subtypes of a class or interface, usually Object, or a different container class has to be created for each contained class. Generics allow compile-time type checking without having to create a large number of container classes, each containing almost identical code.

Class libraries

- Java libraries are the compiled byte codes of source code developed by the JRE implementor to support application development in Java. Examples of these libraries are:
    - The core libraries, which include:
        - Collection libraries that implement data structures such as lists, dictionaries, trees and sets
        - XML Processing (Parsing, Transforming, Validating) libraries
        - Security
        - Internationalization and localization libraries
    - The integration libraries, which allow the application writer to communicate with external systems. These libraries include:
        - The Java Database Connectivity (JDBC) API for database access
        - Java Naming and Directory Interface (JNDI) for lookup and discovery
        - RMI and CORBA for distributed application development
        - JMX for managing and monitoring applications
    - User Interface libraries, which include:
        - The (heavyweight, or native) Abstract Window Toolkit (AWT), which provides GUI components, the means for laying out those components and the means for handling events from those components
        - The (lightweight) Swing libraries, which are built on AWT but provide (non-native) implementations of the AWT widgetry
        - APIs for audio capture, processing, and playback
- A platform dependent implementation of Java virtual machine (JVM) that is the means by which the byte codes of the Java libraries and third party applications are executed
- Plugins, which enable applets to be run in Web browsers
- Java Web Start, which allows Java applications to be efficiently distributed to end users across the Internet
- Licensing and documentation.

Documentation

Javadoc is a comprehensive documentation system, created by Sun Microsystems, used by many Java developers. It provides developers with an organized system for documenting their code. Whereas normal comments in Java and C are set off with /* and */, the multi-line comment tags, Javadoc comments have an extra asterisk at the beginning, so that the tags are /** and */.

Examples

The following is an example of java code commented with simple Javadoc-style comments:

```
/**
 * A program that does useful things.
 */
public class Program {

 /**
  * A main method.
  * @param args The arguments
  */
 public static void main(String[] args) {
  //do stuff
 }

}
```

Sun has defined and supports four editions of Java targeting different application environments and segmented many of its APIs so that they belong to one of the platforms. The platforms are:

- Java Card for smartcards.
- Java Platform, Micro Edition (Java ME) — targeting environments with limited resources.
- Java Platform, Standard Edition (Java SE) — targeting workstation environments.
- Java Platform, Enterprise Edition (Java EE) — targeting large distributed enterprise or Internet environments.

The classes in the Java APIs are organized into separate groups called packages. Each package contains a set of related interfaces, classes and exceptions. Refer to the separate platforms for a description of the packages available.

The set of APIs is controlled by Sun Microsystems in cooperation with others through the Java Community Process program. Companies or individuals participating in this process can influence the design and development of the APIs. This process has been a subject of controversy.

Sun also provided an edition called PersonalJava that has been superseded by later, standards-based Java ME configuration-profile pairings.

Java Basics

Portability

Java programs are portable across operating systems and hardware environments.
Portability is to your advantage because:

- You need only one version of your software to serve a broad market.
- The Internet, in effect, becomes one giant, dynamic library.
- You are no longer limited by your particular computer platform.

Three features make Java String programs portable:

1. **The language**. The Java language is completely specified; all data-type sizes and formats are defined as part of the language. By contrast, C/C++ leaves these "details" up to the compiler implementor, and many C/C++ programs therefore

are not portable.
2. **The library**. The Java class library is available on any machine with a Java runtime system, because a portable program is of no use if you cannot use the same class library on every platform. Window-manager function calls in a Mac application written in C/C++, for example, do not port well to a PC.
3. **The byte code**. The Java runtime system does not compile your source code directly into machine language, an inflexible and nonportable representation of your program. Instead, Java programs are translated into machine-independent byte code. The byte code is easily interpreted and therefore can be executed on any platform having a Java runtime system. (The latest versions of the Netscape Navigator browser, for example, can run applets on virtually any platform).

Security

The Java language is secure in that it is very difficult to write incorrect code or viruses that can corrupt/steal your data, or harm hardware such as hard disks.
There are two main lines of defense:

- Interpreter level:
- No pointer arithmetic
- Garbage collection
- Array bounds checking
- No illegal data conversions

Browser level (applies to applets only):
No local file I/O
Sockets back to host only

- No calls to native methods

Robustness

The Java language is robust. It has several features designed to avoid crashes during program execution, including:

- No pointer arithmetic
- Garbage collection--no bad addresses
- Array and string bounds checking
- No jumping to bad method addresses

Interfaces and exceptions

Java Program Structure

A file containing Java source code is considered a compilation unit. Such a compilation unit contains a set of classes and, optionally, a package definition to group related classes together. Classes contain data and method members that specify the state and behavior of the objects in your program.
Java programs come in two flavors:

- Standalone applications that have no initial context such as a pre-existing main

window
- Applets for WWW programming

The major differences between applications and applets are:
- Applets are not allowed to use file I/O and sockets (other than to the host platform). Applications do not have these restrictions.
- An applet must be a subclass of the Java Applet class. Aplications do not need to subclass any particular class.
- Unlike applets, applications can have menus.
- Unlike applications, applets need to respond to predefined lifecycle messages from the WWW browser in which they're running.

Java Program Execution

The Java byte-code compiler translates a Java source file into machine independent byte code. The byte code for each publicly visible class is placed in a separate file, so that the Java runtime system can easily find it. If your program instantiates an object of class A, for example, the class loader searches the directories listed in your CLASSPATH environment variable for a file called A.class that contains the class definition and byte code for class A.

There is no link phase for Java programs; all linking is done dynamically at

The following diagram shows an example of the Java compilation and execution sequence for a source file named A.java containing public class A and non-public class B:

Java programs are, in effect, distributed applications. You may think of them as a collection of DLLs (dynamically loadable libraries) that are linked on demand at runtime. When you write your own Java applications, you will often integrate your program with already-existing portions of code that reside on other machines.

A Simple Application

Consider the following trivial application that prints "hi there" to standard

```
public class TrivialApplication {
// args[0] is first argument
// args the second
public static void main(String args[]) {
System.out.println("hi there");
}
}
```

The command java TrivialApplication tells the Java runtime system to begin with the class file TrivialApplication.class and to look in that file for a method with the signature:

```
public static void main(String args[]);
```

The main() method will always reside in one of your class files. The Java language does not allow methods outside of class definitions. The class, in effect, creates scoped symbol StartingClassName.main for your main() method.

Applet Execution

An applet is a Java program that runs within a Java-compatible WWW browser or in an appletviewer. To execute your applet, the browser:

- Creates an instance of your applet
- Sends messages to your applet to automatically invoke predefined lifecycle methods

The predefined methods automatically invoked by the runtime system are:

- init(). This method takes the place of the Applet constructor and is only called once during applet creation. Instance variables should be initialized in this method. GUI components such as buttons and scrollbars should be added to the GUI in this method.
- start(). This method is called once after init() and whenever your applet is revisited by your browser, or when you deiconify your browser. This method should be used to start animations and other threads.
- paint(Graphics g). This method is called when the applet drawing area needs to be redrawn. Anything not drawn by contained components must be drawn in this method. Bitmaps, for example, are drawn here, but buttons are not because they handle their own painting.
- stop(). This method is called when you leave an applet or when you iconify your browser. The method should be used to suspend animations and other threads so they do not burden system resources unnecessarily. It is guaranteed to be called before destroy().
- destroy(). This method is called when an applet terminates, for example, when quitting the browser. Final clean-up operations such as freeing up system resources with dispose() should be done here. The dispose() method of Frame removes the menu bar. Therefore, do not forget to call super.dispose() if you override the default behavior.

The basic structure of an applet that uses each of these predefined methods is:

```
import java.applet.Applet;
// include all AWT class definitions
import java.awt.*;
public class AppletTemplate extends Applet {
public void init() {
// create GUI, initialize applet
}
public void start() {
// start threads, animations etc...
}
public void paint(Graphics g) {
// draw things in g
}
public void stop() {
// suspend threads, stop animations etc...
}
public void destroy() {
// free up system resources, stop threads
}
```

}
All you have to do is fill in the appropriate methods to bring your applet to life. If you don't need to use one or more of these predefined methods, simply leave them out of your applet. The applet will ignore messages from the browser attempting to invoke any of these methods that you don't use.

A Simple Applet
The following complete applet displays "Hello, World Wide Web!" in your browser window:

```
import java.applet.Applet;
import java.awt.Graphics;
public class TrivialApplet extends Applet {
public void paint(Graphics g) {
// display a string at 20,20
g.drawString("Hello, World Wide Web!", 20, 20);
}
}
```

An appletviewer may be used instead of a WWW browser to test applets. For example, the output of TrivialApplet on an appletviewer looks like:

HTML/Applet Interface
The HTML applet tag is similar to the HTML img tag, and has the form:

```
<applet code=AppletName.class width=w height=h>
[parameters]
</applet>
```

where the optional parameters are a list of parameter definitions of the form:

```
<param name=n value=v>
```

An example tag with parameter definitions is:

```
<applet code=AppletName.class width=300 height=200>
<param name=p1 value=34>
<param name=p2 value="test">
</applet>
```

where p1 and p2 are user-defined parameters.
The code, width, and height parameters are mandatory. The parameters codebase, alt, archives, align, vspace, and hspace are optional within the <applet> tag itself. Your applet can access any of these parameters by calling:

```
Applet.getParameter("p")
```

which returns the String value of the parameter. For example, the applet:

```
import java.applet.Applet;
public class ParamTest extends Applet {
public void init() {
System.out.println("width is " + getParameter("width"));
System.out.println("p1 is " + getParameter("p1"));
```

```
System.out.println("p2 is " + getParameter("p2"));
}
}
```
prints the following to standard output:
```
width is 300
p1 is 34
p2 is test
```

Comments

Java comments are the same as C++ comments, i.e.,
```
/* C-style block comments */
```
where all text between the opening /* and closing */ is ignored, and
```
// C++ style single-line comments
```
where all text from the opening // to the end of the line is ignored.
Note that these two comments can make a very useful combination. C-style comments (/* ... */) cannot be nested, but can contain C++ style comments. This leads to the interesting observation that if you always use C++-style comments (// ...), you can easily comment out a section of code by surrounding it with C-style comments. So try to use C++ style comments for your "normal" code commentary, and reserve C-style comments for commenting out sections of code.
The Java language also has a document comment:
```
/** document comment */
```
These comments are processed by the javadoc program to generate documentation from your source code. For example,
```
/** This class does blah blah blah */
class Blah {
/** This method does nothing
/**
* This is a multiple line comment.
* The leading * is not placed in documentation.
*/
public void nothing() {;}
}
```

Declarations

A Java variable may refer to an object, an array, or an item of primitive type. Variables are defined using the following simple syntax:
```
TypeName variableName;
```
For example,
```
int a; // defines an integer
int[] b; // defines a reference to array of ints
Vector v; // reference to a Vector object
```

Primitive Types

The Java language has the following primitive types:

Primitive Types

Primitive Type Description

boolean true/false

byte 8 bits

char 16 bits (UNICODE)

short 16 bits

int 32 bits

long 64 bits

float 32 bits IEEE 754-1985

double 64 bits IEEE 754-1985

Java int types may **not** be used as boolean types and are always signed.

Objects

A simple C++ object or C struct definition such as "Button b;" allocates memory on the stack for a Button object and makes b refer to it. By contrast, you must specifically instantiate Java objects with the new operator. For example,

```
// Java code
void foo() {
// define a reference to a Button; init to null
Button b;
// allocate space for a Button, b points to it
b = new Button("OK");
int i = 2;
}
```

As the accompanying figure shows, this code places a reference b to the Button object on the stack and allocates memory for the new object on the heap.

The equivalent C++ and C statements that would allocate memory on the heap would be:

```
// C++ code
Button *b = NULL; // declare a new Button pointer
b = new Button("OK"); // point it to a new Button
/* C code */
Button *b = NULL; /* declare a new Button pointer */
b = calloc(1, sizeof(Button)); /* allocate space for a Button */
init(b, "OK"); /* something like this to init b */
```

All Java objects reside on the heap; there are no objects stored on the stack. Storing objects on the heap does not cause potential memory leakage problems because of garbage collection.

Each Java primitive type has an equivalent object type, e.g., Integer, Byte, Float, Double. These primitive types are provided in addition to object types purely for efficiency. An int is much more efficient than an Integer.

Strings

Java string literals look the same as those in C/C++, but Java strings are real objects, not pointers to memory. Java strings may or may not be null-terminated. Every string literal such as
"a string literal"
is interpreted by the Java compiler as
new String("a string literal")
Java strings are constant in length and content. For variable-length strings, use StringBuffer objects.
Strings may be concatenated by using the plus operator:
String s = "one" + "two"; // s == "onetwo"
You may concatenate any object to a string. You use the toString() method to convert objects to a String, and primitive types are converted by the compiler. For example,
String s = "1+1=" + 2; // s == "1+1=2"
The length of a string may be obtained with String method length(); e.g., "abc".length() has the value 3.
To convert an int to a String, use:
String s = String.valueOf(4);
To convert a String to an int, use:
int a = Integer.parseInt("4");

Array Objects
In C and C++, arrays are pointers to data in memory. Java arrays are objects that know the number and type of their elements. The first element is index 0, as in C/C++.
Generic Array Object
elements
element type
element 0
element 1
...
element n-1
The syntax for creating an array object is:
TypeName[] variableName;
This declaration defines the array object--it does not allocate memory for the array object nor does it allocate the elements of the array In addition, you may not specify a size within the square brackets.
To allocate an array, use the new operator:
int[] a = new int; // Java code: make array of 5 ints
new int

5
int
0
0
0
0
0
In C or C++, by contrast, you would write either
/* C/C++ code: make array of 5 ints on the stack */
int a;
or
/* C/C++ code: make array of 5 ints on the heap */
int *a = new int;
An array of Java objects such as
// Java code: make array of 5 references to Buttons
Button[] a = new Button;
creates the array object itself, but not the elements:
new Button
5
Button
null pointer
null pointer
null pointer
null pointer
null pointer
You must use the new operator to create the elements:
a[0] = new Button("OK");
a = new Button("QUIT");
In C++, to make an array of pointers to objects you would write:
// C++: make an array of 5 pointers to Buttons
Button **a = new Button *; // Create the array
a[0] = new Button("OK"); // create two new buttons
a = new Button("QUIT");
In C, code for the same task would look like:
/* C: make an array of 5 pointers to structs */
/* Allocate the array */
Button **a = calloc(5, sizeof(Button *));
/* Allocate one button */
a[0] = calloc(1, sizeof(Button));
/* Init the first button */
setTitle(a[0], "OK");
/* Allocate another button */
a = calloc(1, sizeof(Button));
/* Init the second button */

setTitle(a, "QUIT");
Multi-dimensional Java arrays are created by making arrays of arrays, just as in C/C++. For example,
T[][] t = new T;
makes a five-element array of ten arrays of references to objects of type T. This statement does not allocate memory for any T objects.
Accessing an undefined array element causes a runtime exception called ArrayIndexOutOfBoundsException.
Accessing a defined array element that has not yet been assigned to an object results in a runtime NullPointerException.

Initializers

Variables may be initialized as follows:

- Primitive types

int i = 3;
boolean g = true;

- Objects

Button b = null;
Employee e = new Employee();

- Arrays

int[] i = {1, 2, 3, 4};
or in Java 1.1
int[] i;
i = new int[] {1, 2, 3, 4};

Constants

Variables modified by the static final keywords are constants (equivalent to the const keyword in C++; no equivalent in C). For example,
// same as "const int version=1;" in C++
static final int version = 1;
static final String Owner = "Terence";

Expressions

Most Java expressions are similar to those in C/C++.

Constant Expressions

Item Examples or Description

id i, nameList
qualified-id Integer.MAX_VALUE, obj.member,
npackage.class, package.obj
id[e][f]...[g] a[i], b
String literal "Jim", delimited by ""
char literal 'a', '\t', delimited by '
Unicode character constant \u00ae
boolean literal true, false (not an int)
int constant 4
float constant 3.14f, 2.7e6F, f or F suffix
double constant 3.14, 2.7e6D, (default) / d or D suffix
hexadecimal constant 0x123
octal constant 077

null the null object (note lowercase!)
this the current object
super the superclass view of this object

General Expressions

Item Examples or Description

id i, nameList
obj.method(args) instance method call
class.method(args) class method call
(expr) (3+4)*7
new T(constructor-args) instantiates a new object or class T
new T[e][f]...[g] allocates an array object

Operators

The Java language has added the >>> zero-extend right-shift operator to the set of C++ operators. (C++ operators include instanceof and new, which are not present in C. Note that sizeof has been removed, as memory allocation is handled for you.) The operators, in order of highest to lowest priority, are:

- new
- .
- -- ++ + - ~ ! (TypeName)
- * / %
- + -
- << >> >>>
- < > <= >= instanceof
- == !=
- &
- ^
- |
- &&
- ||
- ?:
- = *= /= %= += -= <<= >>= >>>= &= ^= |=

Note that the precedence of the new operator and the '.' operator bind differently than in C++. A proper Java statement is:

```
// Java code
new T().method();
```

In C++, you would use:

```
// C++ code
(new T)->method();
```

Statements

Java statements are similar to those in C/C++ as the following table shows.

Forms of Common Statements

Statement Examples
if if (boolean-expr) stat1
if (boolean-expr) stat1 else stat2
switch
switch (int-expr) {

```
case int-const-expr : stat1
case int-const-expr : stat2
default : stat3
}
for for (int i=0; i<10; i++) stat
while while (boolean-expr) stat
do-while do { stats } while (boolean-expr)
return return expr;
```

The Java break and continue statements may have labels. These labels refer to the specific loop that the break or continue apply to. (Each loop can be preceded by a label.)

Java Semantics

We say that the Java language has "reference semantics" and C/C++ have "copy semantics." This means that Java objects are passed to methods by reference in Java, while objects are passed by value in C/C++.

Java primitive types, however, are not treated in the same way as Java objects. Primitive types are assigned, compared, and passed as arguments using copy

semantics, just as in C/C++. For example, i = j for two int variables i and j performs a 32-bit integer copy.

Assignment of Objects

Assignment makes two variables refer to the same object. For example,

```
class Data {
public int data = 0;
public Data(int d) { data = d; }
}
I Data a = new Data(1); // a.data is 1
I Data b = new Data(2); // b.data is 2
II b = a; // b.data and a.data are 1
III a.data = 3; // b.data and a.data are 3
IV a = new Data(4); // b.data is 3, a.data is 4
```

To copy objects, define and use clone():

```
class Data implements Cloneable {
public int data = 0;
public Data(int d) { data = d; }
public Object clone() {
Data d = (Data) super.clone();
d.data = data;
return d;
}
}
...
Data a = new Data(1); // a.data is 1
Data b = new Data(2); // b.data is 2
b = a.clone(); // b.data and a.data are 1
```

```
a.data = 3; // b.data is 1, a.data is 3
```
Note: The above class definition requires exception handling code. We, however, have not yet discussed exception handling. For now, pretend that it is not necessary.

Method Parameters and Return Values

Arguments and return values for primitive types are passed by value to and from all Java methods because they are implied assignments, as in C/C++. However, all Java objects are passed by reference. For example, the C/C++ code:

```
// C++ code
int foo(int j) { return j + 34;}
Button *bfoo(Button *b) {
if ( b != NULL ) return b;
else return new Button();
}
```
or, in C
```
/* C code */
int foo(int j) { return j + 34;}
Button *bfoo(Button *b) {
if ( b != NULL ) return b;
else return calloc(sizeof(Button));
}
```
would be written in the Java language:
```
// Java code
int foo(int j) { return j + 34;}
Button bfoo(Button b) {
if ( b != null ) return b;
else return new Button("OK");
}
```
Equality

Two Java primitive types are equal (using the == operator) when they have the same value (e.g., "3 == 3"). However, two object variables are equal if and only if they refer to the same instantiated object--a "shallow" comparison. For example,

```
void test() {
Data a = new Data(1);
Data b = new Data(2);
Data c = new Data(1);
// a == b is FALSE
// a == c is FALSE (in C++, this'd be TRUE)
```


```
Data d = a;
Data e = a;
// d == e is TRUE,
// d,e are referring to same object
}
```
To perform a "deep" comparison, the convention is to define a method called

equals(). You would rewrite Data as:

```
class Data {
public int data = 0;
public Data(int d) { data = d; }
boolean equals(Data d) {
return data == d.data;
}
}
...
Data a = new Data(1);
Data b = new Data(1);
// a.equals(b) is true!!!!
```

No Pointers!

The Java language does not have pointer types nor address arithmetic. Java variables are either primitive types or references to objects. To illustrate the difference between C/C++ and Java semantics, consider the following equivalent code fragments.

```
// C++ code (C code would be similar)
Stack *s = new Stack; // point to a new Stack
s->push(...);
// dereference and access method push()
```

The equivalent Java code is:

```
// Java code
// internally, consider s to be a (Stack *)
Stack s = new Stack();
// dereference s automatically
.push(...);
```

Garbage Collection

An automatic garbage collector deallocates memory for objects that are no longer needed by your program, thereby relieving you from the tedious and error-prone task of deallocating your own memory.

As a consequence of automatic garbage collection and lack of pointers, a Java object is either null or valid--there is no way to refer to an invalid or stale object (one that has been deallocated).

To illustrate the effect of a garbage collector, consider the following C++ function that allocates 1000 objects on the heap via the new operator (a similar C function would allocate memory using calloc/malloc):

```
// C++ code
void f() {
T *t;
for (int i = 1; i <= 1000; i++) {
t = new T; // ack!!!!!
}
```

```
}
```
Every time the loop body is executed, a new instance of class T is instantiated, and t is pointed to it. But what happens to the instance that t used to point to? It's still allocated, but nothing points to it and therefore it's inaccessible. Memory in this state is referred to as "leaked" memory.
In the Java language, memory leaks are not an issue. The following Java method causes no ill effects:

```
// Java code
void f() {
T t;
for (int i = 1; i <= 1000; i++) {
t = new T();
}
}
```

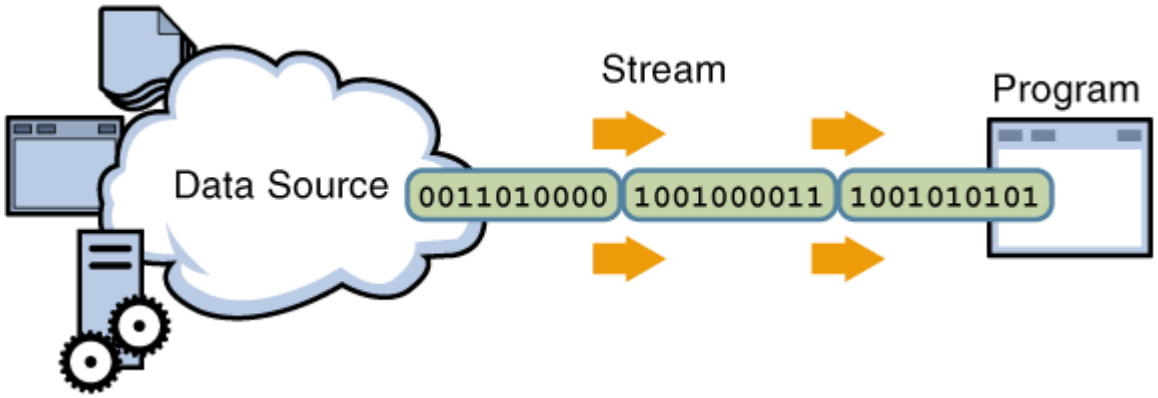
In Java, each time t is assigned a new reference, the old reference is now available for garbage collection. Note that it isn't immediately freed; it remains allocated until the garbage collector thread is next executed and notices that it can be freed. Put simply, automatic garbage collection reduces programming effort,

I/O Streams

An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

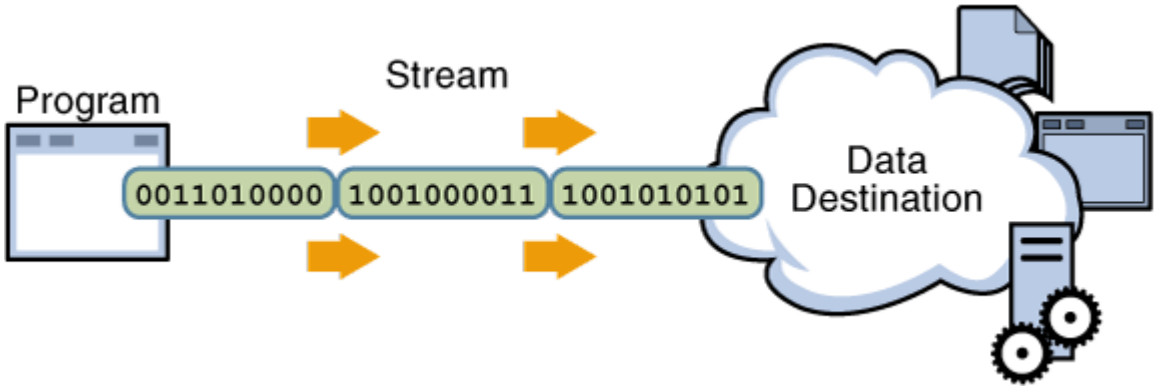
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an input stream to read data from a source, one item at a time:



Reading information into a program.

A program uses an output stream to write data to a destination, one item at time:



Writing information from a program.

In this lesson, we'll see streams that can handle all kinds of data, from primitive values to advanced objects.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

In the next section, we'll use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O. For sample input, we'll use the example file xanadu.txt, which contains the following verse:

In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

File I/O

The java.nio.file package and its related package, java.nio.file.attribute, provide comprehensive support for file I/O and for accessing the file system. Though the API has many classes, you need to focus on only a few entrypoints — you will see that this API is very intuitive and easy to use.

The tutorial starts by asking what is a Path? Then the Path class, the primary entrypoint for the package, is introduced. Methods in the Path class relating to syntactic operations are explained. The tutorial then moves on to Path methods that deal with file operations.

First, some concepts common to many file operations are introduced. The tutorial then covers methods for checking, deleting, copying, and moving files.

The tutorial shows how metadata is managed, before moving on to file I/O and directory I/O. Random access files are explained and issues specific to symbolic and hard links are examined.

Next, some of the very powerful, but more advanced, topics are covered. First, the ability to recursively walk the file tree is demonstrated, followed by information on how to search for files using wild cards. Next, how to watch a directory for changes is explained and demonstrated. Then, methods that didn't fit elsewhere are given some attention.

Finally, if you have file I/O code written prior to JDK7, there is a map from the old API to the new API, as well as important information about the File.toPath method for those who would like to leverage the new API without re-writing existing code.

Class InternetAddress

java.lang.Object
  └javax.mail.Address
    └**javax.mail.internet.InternetAddress**
All Implemented Interfaces:
Cloneable, Serializable
public class **InternetAddress**
extends Address
implements Cloneable

This class represents an Internet email address using the syntax of RFC822. Typical address syntax is of the form "user@host.domain" or "Personal Name ".

Constructor Summary
InternetAddress() Default constructor.
InternetAddress(String address) Constructor.
InternetAddress(String address, boolean strict)

Parse the given string and create an InternetAddress.
InternetAddress(String address, String personal) Construct an InternetAddress given the address and personal name.
InternetAddress(String address, String personal, String charset) Construct an InternetAddress given the address and personal name.

Method Summary	
Object	**clone**() Return a copy of this InternetAddress object.
boolean	**equals**(Object a) The equality operator.
String	**getAddress**() Get the email address.
InternetAddress[]	**getGroup**(boolean strict) Return the members of a group address.
static InternetAddress	**getLocalAddress**(Session session) Return an InternetAddress object representing the current user.
String	**getPersonal**() Get the personal name.
String	**getType**() Return the type of this address.
int	**hashCode**() Compute a hash code for the address.
boolean	**isGroup**() Indicates whether this address is an RFC 822 group address.
static InternetAddress[]	**parse**(String addresslist) Parse the given comma separated sequence of addresses into InternetAddress objects.
static InternetAddress[]	**parse**(String addresslist, boolean strict) Parse the given sequence of addresses into InternetAddress objects.
static InternetAddress[]	**parseHeader**(String addresslist, boolean strict) Parse the given sequence of addresses into InternetAddress objects.
void	**setAddress**(String address)

	Set the email address.
void	**setPersonal**(String name) Set the personal name.
void	**setPersonal**(String name, String charset) Set the personal name.
String	**toString**() Convert this address into a RFC 822 / RFC 2047 encoded address.
static String	**toString**(Address[] addresses) Convert the given array of InternetAddress objects into a comma separated sequence of address strings.
static String	**toString**(Address[] addresses, int used) Convert the given array of InternetAddress objects into a comma separated sequence of address strings.
String	**toUnicodeString**() Returns a properly formatted address (RFC 822 syntax) of Unicode characters.
void	**validate**() Validate that this address conforms to the syntax rules of RFC 822.

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Field Detail

address

protected String **address**

personal

protected String **personal**
The personal name.

encodedPersonal

protected String **encodedPersonal**

The RFC 2047 encoded version of the personal name.

This field and the personal field track each other, so if a subclass sets one of these fields directly, it should set the other to null, so that it is suitably recomputed.

Method Detail

clone

public Object **clone**()

Return a copy of this InternetAddress object.

Since:

JavaMail 1.2

getType

public String **getType**()

Return the type of this address. The type of an InternetAddress is "rfc822".

Specified by:

getType in class Address

Returns:

address type

See Also:

InternetAddress

setAddress

public void **setAddress**(String address)

Set the email address.

Parameters:

address - email address

---

setPersonal

public void **setPersonal**(String name,
String charset)
throws UnsupportedEncodingException

Set the personal name. If the name contains non US-ASCII characters, then the name will be encoded using the specified charset as per RFC 2047. If the name contains only US-ASCII characters, no encoding is done and the name is used as is.

Parameters:

name - personal name

charset - MIME charset to be used to encode the name as per RFC 2047

Throws:
UnsupportedEncodingException - if the charset encoding fails.
See Also:
setPersonal(String)

setPersonal

public void **setPersonal**(String name)
throws UnsupportedEncodingException
Set the personal name. If the name contains non US-ASCII characters, then the name will be encoded using the platform's default charset. If the name contains only US-ASCII characters, no encoding is done and the name is used as is.
Parameters:
name - personal name
Throws:
UnsupportedEncodingException - if the charset encoding fails.
See Also:
setPersonal(String name, String charset)

getAddress

public String **getAddress**()
Get the email address.
Returns:
email address

getPersonal

public String **getPersonal**()
Get the personal name. If the name is encoded as per RFC 2047, it is decoded and converted into Unicode. If the decoding or conversion fails, the raw data is returned as is.
Returns:
personal name

toString

public String **toString**()
Convert this address into a RFC 822 / RFC 2047 encoded address. The resulting string contains only US-ASCII characters, and hence is mail-safe.
Specified by:
toString in class Address
Returns:
possibly encoded address string

toUnicodeString

public String **toUnicodeString**()

Returns a properly formatted address (RFC 822 syntax) of Unicode characters.

Returns:

Unicode address string

Since:

JavaMail 1.2

equals

public boolean **equals**(Object a)

The equality operator.

Specified by:

equals in class Address

Parameters:

a - Address object

hashCode

public int **hashCode**()

Compute a hash code for the address.

toString

public static String **toString**(Address[] addresses)

Convert the given array of InternetAddress objects into a comma separated sequence of address strings. The resulting string contains only US-ASCII characters, and hence is mail-safe.

Parameters:

addresses - array of InternetAddress objects

Returns:

comma separated string of addresses

Throws:

ClassCastException, - if any address object in the given array is not an InternetAddress object. Note that this is a RuntimeException.

---

toString

public static String **toString**(Address[] addresses,
int used)

Convert the given array of InternetAddress objects into a comma separated sequence of address strings. The resulting string contains only US-ASCII characters, and hence is mail-safe.

The 'used' parameter specifies the number of character positions already taken up in the field into which the resulting address sequence string is to be inserted. It is used to determine the line-break positions in the resulting address sequence string.

Parameters:
addresses - array of InternetAddress objects
used - number of character positions already used, in the field into which the address string is to be inserted.
Returns:
comma separated string of addresses
Throws:
ClassCastException, - if any address object in the given array is not an InternetAddress object. Note that this is a RuntimeException.

---

getLocalAddress

public static InternetAddress **getLocalAddress**(Session session)

Return an InternetAddress object representing the current user. The entire email address may be specified in the "mail.from" property. If not set, the "mail.user" and "mail.host" properties are tried. If those are not set, the "user.name" property and InetAddress.getLocalHost method are tried. Security exceptions that may occur while accessing this information are ignored. If it is not possible to determine an email address, null is returned.
Parameters:
session - Session object used for property lookup
Returns:
current user's email address

---

parse

public static InternetAddress[] **parse**(String addresslist)
throws AddressException

Parse the given comma separated sequence of addresses into InternetAddress objects. Addresses must follow RFC822 syntax.
Parameters:
addresslist - comma separated address strings
Returns:
array of InternetAddress objects
Throws:
AddressException - if the parse failed

---

parse

public static InternetAddress[] **parse**(String addresslist,
boolean strict)
throws AddressException

Parse the given sequence of addresses into InternetAddress objects. If strict is false, simple email addresses separated by spaces are also allowed. If strict is true, many (but not all) of the RFC822 syntax rules are enforced. In particular, even if strict is true, addresses composed of simple names (with no "@domain" part) are allowed. Such "illegal" addresses are not uncommon in real messages.

Non-strict parsing is typically used when parsing a list of mail addresses entered by a human. Strict parsing is typically used when parsing address headers in mail messages.

Parameters:
addresslist - comma separated address strings
strict - enforce RFC822 syntax
Returns:
array of InternetAddress objects
Throws:
AddressException - if the parse failed

---

parseHeader

public static InternetAddress[] **parseHeader**(String addresslist,
boolean strict)
throws AddressException

Parse the given sequence of addresses into InternetAddress objects. If strict is false, the full syntax rules for individual addresses are not enforced. If strict is true, many (but not all) of the RFC822 syntax rules are enforced.

To better support the range of "invalid" addresses seen in real messages, this method enforces fewer syntax rules than the parse method when the strict flag is false and enforces more rules when the strict flag is true. If the strict flag is false and the parse is successful in separating out an email address or addresses, the syntax of the addresses themselves is not checked.

Parameters:
addresslist - comma separated address strings
strict - enforce RFC822 syntax
Returns:
array of InternetAddress objects
Throws:
AddressException - if the parse failed
Since:
JavaMail 1.3

---

validate

public void **validate**()
throws AddressException
Validate that this address conforms to the syntax rules of RFC 822. The current implementation checks many, but not all, syntax rules. Note that even though the syntax of the address may be correct, there's no guarantee that a mailbox of that name exists.
Throws:
AddressException - if the address isn't valid.
Since:
JavaMail 1.3

---

isGroup

public boolean **isGroup**()
Indicates whether this address is an RFC 822 group address. Note that a group address is different than the mailing list addresses supported by most mail servers. Group addresses are rarely used; see RFC 822 for details.
Returns:
true if this address represents a group
Since:
JavaMail 1.3

---

getGroup

public InternetAddress[] **getGroup**(boolean strict)
throws AddressException
Return the members of a group address. A group may have zero, one, or more members. If this address is not a group, null is returned. The strict parameter controls whether the group list is parsed using strict RFC 822 rules or not. The parsing is done using the parseHeader method.
Returns:

array of InternetAddress objects, or null
Throws:
AddressException - if the group list can't be parsed
Since:
JavaMail 1.3

Socket Programming

Abstract: A Basic Socket Processing How To. Learn to build a simple socket client, a socket server that handles one connection at a time, and a socket server that can handle mulitple socket connections.

Introduction

One of the most basic network programming tasks you'll likely face as a Java programmer is performing socket functions. You may have to create a network client that talks to a server via a socket connection. Or, you may have to create a server that listens for socket connections. Either way, sooner or later you're going to deal with sockets. What is a socket you ask? Think of a socket as the basic communication interface between networked computers. Sockets allow you the programmer to treat the network connection as you would any other I/O. In Java, sockets are the lowest level of network coding.

During the next few paragraphs, we'll work through some examples of socket programming in Java: a simple client, a simple server that takes one connection at a time, and a server that allows multiple socket connections.

SocketClient: A Simple TCP/IP Socket Client

```
package bdn;
/*  The java.net package contains the basics needed for network operations. */
import java.net.*;
/* The java.io package contains the basics needed for IO operations. */
import java.io.*;
/** The SocketClient class is a simple example of a TCP/IP Socket Client.
 *
 */

public class SocketClient {
```

Let's start by creating a class called SocketClient. We'll put this into a package called bdn. The only packages that we're going to need in this example are java.net and java.io. If you've not dealt with the java.net. package before, as it's name implies, it contains the basic classes and methods you'll need for network programming (see your JBuilder help files orOne of the cool things about java is the consistent use of InputStreams and

OutputStreams to read and write I/O, regardless of the device. In other words, you can almost always be assured that if you are reading from any input source, you will use an InputStream...when writing to output sources you'll use an OutputStream. This means reading and writing across a network is almost the same as reading and writing files. For this reason, we need import that java.io package into our program.

Some House Keeping

```
public static void main(String[] args) {
  /** Define a host server */
  String host = "localhost";
  /** Define a port */
  int port = 19999;

  StringBuffer instr = new StringBuffer();
  String TimeStamp;
  System.out.println("SocketClient initialized");
```

In order to make a socket connection, you need to know a couple of pieces of information. First you need a host to connect to. In this example we're going to be running the client(s) and the server on the same machine. We define a String host as **localhost**.

Note: we could have used the TCP/IP address **127.0.0.1** instead of **localhost**.

The next piece of information we need to know is the TCP/IP port that the program is going to be communicating on. TCP/IP uses ports because it is assumed that servers will be doing more than one network function at a time and ports provide a way to maange this. For example: a server may be serving up web pages (port 80), it may have a FTP (port 21) server, and it may be handling a SMTP mail server (port 25). Ports are assigned by the server. The client needs to know what port to use for a particular service. In the TCP/IP world, each computer with a TCP/IP address has access to 65,535 ports. Keep in mind that ports are not physical devices like a serial, parallel, or USB port. They are an abstraction in the computer's memory running under the TCP/IP transport layer protocol.

Note: Ports 1 – 1023 are reserved for services such as HTTP, FTP, email, and Telnet.

Now back to the code. We create an int called **port**. The server we're going to build later in the article will be listening on port 19999. As a result we initialize **port** to 19999.

A couple of other items that we define here are a StringBuffer **instr** to be used for reading our InputStream. We also define a String **TimeStamp** that we'll use to communicate with the server. Lastly, we System.out.println() a message to let us know the program has begun…this kind of stuff is certainly not necessary, but I've found occasionally logging a program status message gives people a peace of mind that a program's actually doing something.

Requesting a Socket and Establishing a Connection

```
try {
  /** Obtain an address object of the server */
  InetAddress address = InetAddress.getByName(host);
  /** Establish a socket connetion */
  Socket connection = new Socket(address, port);
  /** Instantiate a BufferedOutputStream object */
```

Now we create a try-catch block. This block is needed because the methods of several classes we're going to reference here throw exceptions. In our example, we're primarily concerned with IOExceptions, and will specifically capture that one (in real world situations we'd want to deal with this exception more thoroughly). All other exceptions will be captured with a generic Exception.

Note: You should spend some time reviewing the various javadocs on the classes and methods you use. There are often specific exceptions that you want to catch and deal with. Example: had we built an applet that allowed a person to enter the servers and ports they wanted to connect to, we would have wanted to deal with UnknownHostException in the event they keyed an invalid host.

In order to establish a connection with a server, we must first obtain the server's 32-bit IP address. We obtain the IP address by invoking the InetAddress.getByName() method. As it describes, we pass this method the name of the host we're looking to connect to. It returns an InetAddress object address containing the host name/IP address pair (i.e. Using localhost in the getByName() method will return localhost/127.0.0.1 in the InetAddress object).

Once we've obtained the InetAddress object, we're ready to establish a socket connection with our server. We create a Socket called **connection** by instantiating a new Socket object with the InetAdress object address and our previously created int **port**. If the server is not responding on the port we're looking for, we'll get a "java.netConnectException: Connection refused:.." message.

We've established our connection. Now we want to write some information to the server. As mentioned previously, Java treats reading and writing sockets is much like reading and writing files. Subsequently we start by establishing an OutputStream object. In general TCP stacks use buffers to improve performance within the network. And, although it's not necessary, we might as well use BufferedInputStreams and BufferedOutputStreams when reading and writing data across the network. We instantiate a BufferedOutputStream object **bos** by requesting an OutputStream from our socket connection.

Writing to the Socket

```
BufferedOutputStream bos = new BufferedOutputStream(connection.
```

```
    getOutputStream());

/** Instantiate an OutputStreamWriter object with the optional character
 * encoding.
 */
OutputStreamWriter osw = new OutputStreamWriter(bos, "US-ASCII");
```

We could use the BufferedOutputStream.write() method to write bytes across the socket. I prefer to use OutputStreamWriter objects to write on because I'm usually dealing in multiple platforms and like to control the character encoding. Also, with OutputStreamWriter you can pass objects such as Strings without converting to byte, byte arrays, or int values…ok I'm lazy…so what.

Note: If you don't handle the character encoding and are reading and writing to an IBM mainframe from a Windows platform, you'll probably end up with garbage because IBM mainframes tend to encode characters as EBCDIC and Windows encodes characters as ASCII.

We create an OutputStreamWriter **osw** by instantiating it with our BufferedOutputStream **bos** and optionally the character encoding US-ASCII.

```
TimeStamp = new java.util.Date().toString();
String process = "Calling the Socket Server on "+ host + " port " + port +
   " at " + TimeStamp +  (char) 13;

/** Write across the socket connection and flush the buffer */
osw.write(process);
osw.flush();
```

As shown above, we're creating two Strings TimeStamp and process to be written to the server. We call the osw.write() method from our OutputStreamWriter object, passing the String process to it. Please note that we placed a char(13) at the end of process...we'll use this to let the server know we're at the end of the data we're sending. The last item we need to take care of is flushing the buffer. If we don't do this, then we can't guarantee that the data will be written across the socket in a timely manner.

Reading from the Socket

```
/** Instantiate a BufferedInputStream object for reading
/** Instantiate a BufferedInputStream object for reading
 * incoming socket streams.
 */
```

```
    BufferedInputStream bis = new BufferedInputStream(connection.
       getInputStream());
    /**Instantiate an InputStreamReader with the optional
     * character encoding.
     */

    InputStreamReader isr = new InputStreamReader(bis, "US-ASCII");

    /**Read the socket's InputStream and append to a StringBuffer */
    int c;
    while ( (c = isr.read()) != 13)
      instr.append( (char) c);

    /** Close the socket connection. */
    connection.close();
    System.out.println(instr);
    }
   catch (IOException f) {
    System.out.println("IOException: " + f);
   }
   catch (Exception g) {
    System.out.println("Exception: " + g);
   }
  }
}
```

The last thing we want to do is read the server's response. As mentioned before, most networks buffer socket traffic to improve performance. For this reason we're using the BufferedInputStream class. We start by instantiating a BufferInputStream object **bis**, calling the getInputStream() method of our Socket object **connection**. We instantiate an InputStreamReader object **isr**, passing our BufferedInputStream object **bis** and an optional character encoding of US-ASCII.

We create an int **c** that will be used for reading bytes from the BufferedInputStream. We create a while…loop reading bytes and stuffing them into a StringBuffer object **instr** until we encounter a char(13), signaling the end of our stream. Once we've read the socket, we close the socket and process the information…in this example we're just going to send it to the console. The last thing we do is create code in our catch block to deal with exceptions.

Now that we've created the client, what do we do with it? Compiling and running it will give us the following message:

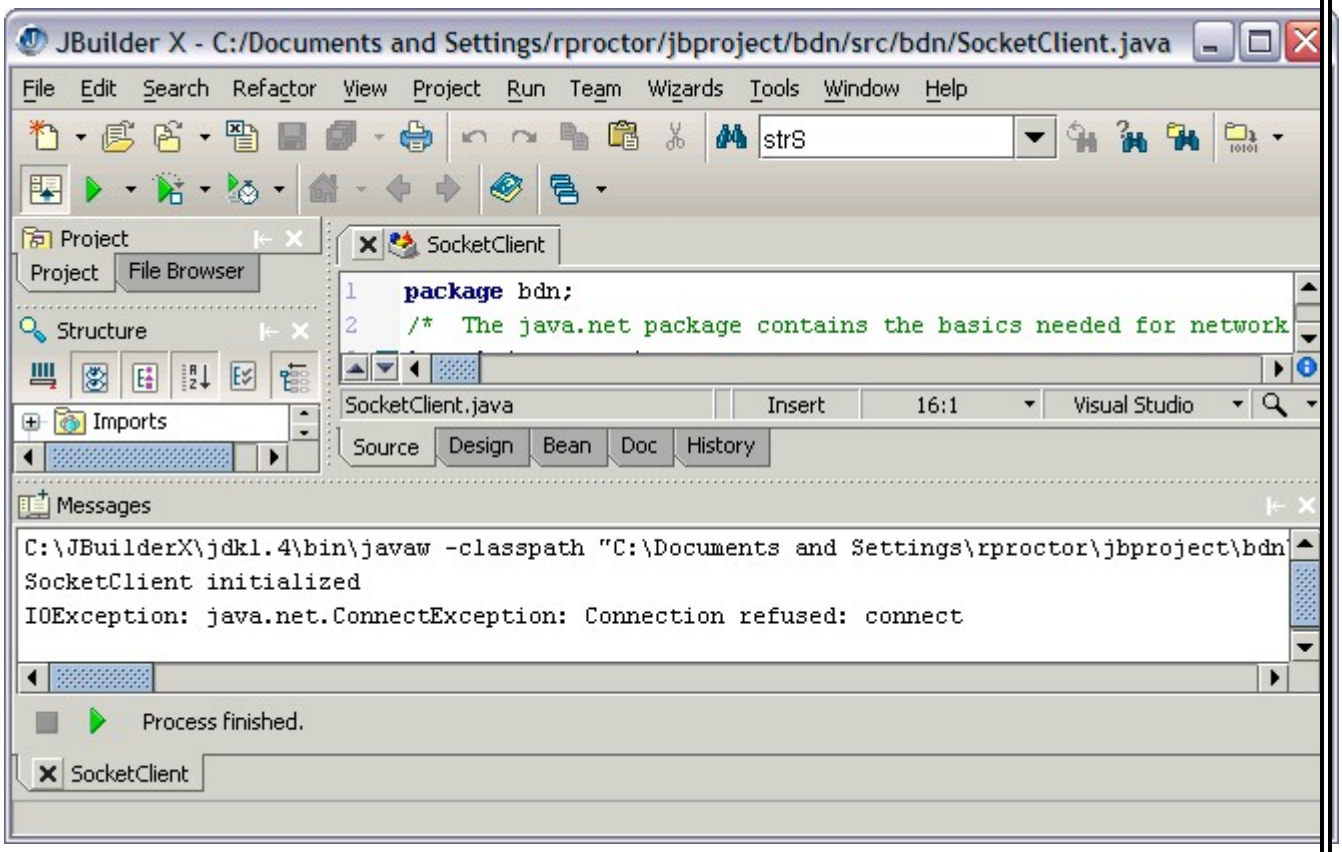


Figure 1: SocketClient Running without Server

We get this message because we don't have a server listening on port 19999 and therefore can't establish a connection with it.

Now it's time to create the server…

SingleSocketServer: A Server That Process One Socket at a Time

```
package bdn;

import java.net.*;
import java.io.*;
import java.util.*;

public class SingleSocketServer {
```

```
static ServerSocket socket1;
protected final static int port = 19999;
static Socket connection;

static boolean first;
static StringBuffer process;
static String TimeStamp;
```

We start by importing the same packages we did with our SocketClient class. We set up a few variables. Of note, this time we're setting up a ServerSocket object called **socket1**. As its name implies, we use the ServerSocket class to set up a new server. As we'll see later, ServerSockets can be created to listen on a particular port and accept and deal with incoming sockets. Depending on the type of server we build, we can also process InputStreams and OutputStreams.

```
public static void main(String[] args) {
  try{
    socket1 = new ServerSocket(port);
    System.out.println("SingleSocketServer Initialized");
    int character;
```

In the main() method, we start with a try…catch block. Next we instantiate a new ServerSocket object **socket1** using the port value of 19999…the same port that we were looking to connect to with our SocketClient class. Finally, we send a message to the console to let the world know we're running.

```
    while (true) {
      connection = socket1.accept();

      BufferedInputStream is = new BufferedInputStream(connection.getInputStream());
      InputStreamReader isr = new InputStreamReader(is);
      process = new StringBuffer();
      while((character = isr.read()) != 13) {
        process.append((char)character);
      }
      System.out.println(process);
      //need to wait 10 seconds for the app to update database
      try {
        Thread.sleep(10000);
      }
```

```
        catch (Exception e){}
        TimeStamp = new java.util.Date().toString();
        String returnCode = "SingleSocketServer repsonded at "+ TimeStamp + (char) 13;
                                                BufferedOutputStream os = new
BufferedOutputStream(connection.getOutputStream());
        OutputStreamWriter osw = new OutputStreamWriter(os, "US-ASCII");
        osw.write(returnCode);
        osw.flush();
      }
     }
     catch (IOException e) {}
      try {
        connection.close();
      }
      catch (IOException e) {}
   }
}
```

Since we're running a server, we can assume that it's always ready to accept and process socket connections. Using a while(true)…loop helps us accomplish this task. Within this block of code, we start with the ServerSocket **socket1's** accept() method. The accept() method basically stops the flow of the program and waits for an incoming socket connection. When a client connects, our Socket object **connection** is instantiated. Once this is done, our program continues through the code.

Once a Connection is Made

We start by processing the InputStream coming from our socket. (For details on this see the explanation for InputStreams in the Reading from the Socket section of this article).

Note: I've thrown a line of code that says Thread.sleep(10000). All I'm doing here is putting the current thread to sleep for 10 seconds. I added this piece of code is purely for the purpose of demonstrating socket connections. It would not be used in a real-world server application.

After we've processed the information in the incoming socket, we want to return some information, in the form of an OutputStream, back to the client. The process here is the same process we used in the SocketClient class (see the section entitled: Writing to the Socket). Once the OutputStream is written, the server is now ready to accept another socket.

Running The SingleSocketServer and SocketClient Classes Together

Let's look at what happens when we run SingleSocketServer and SocketClient together. Executing the programs in JBuilder X will look like this:

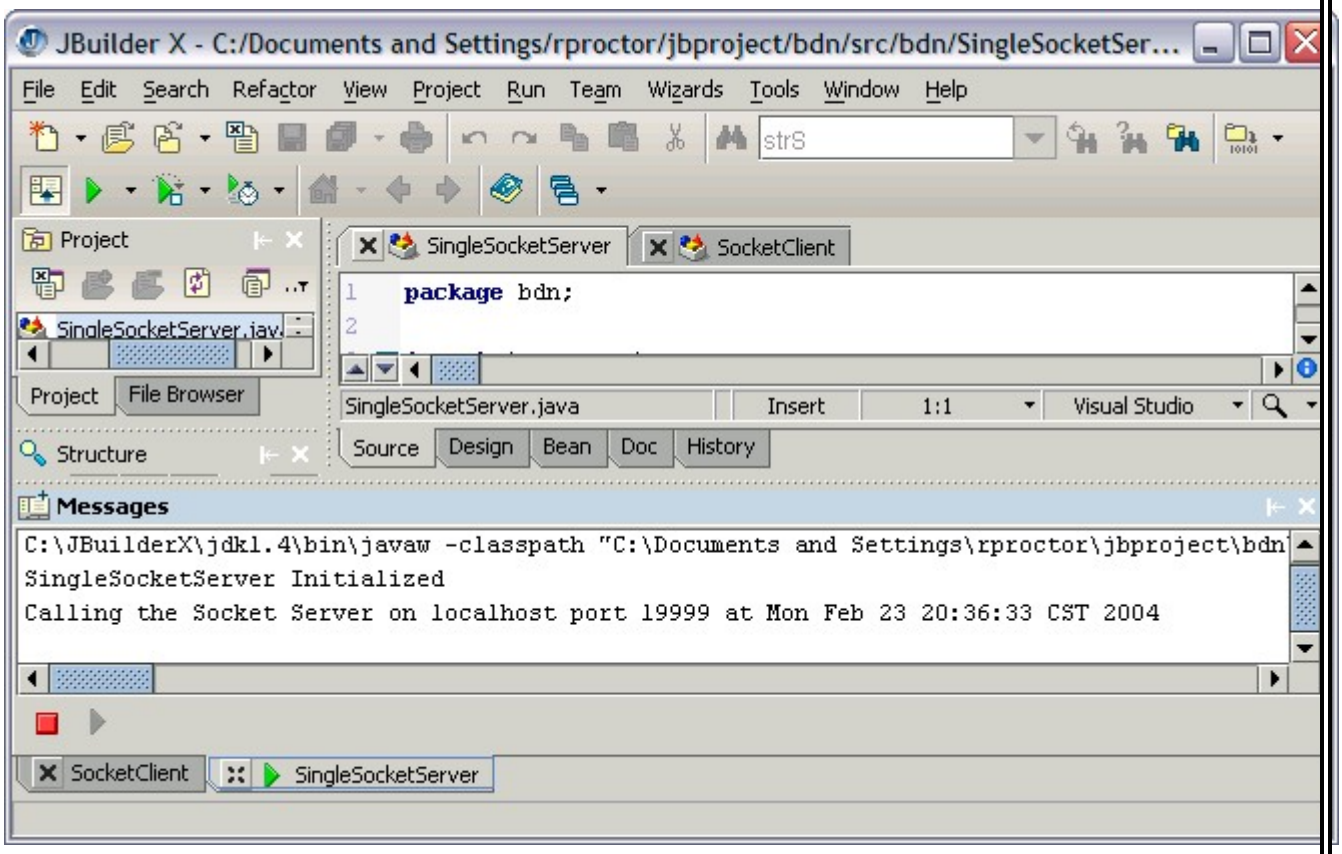


Figure 2: SingleSocketServer After One SocketClient Connection

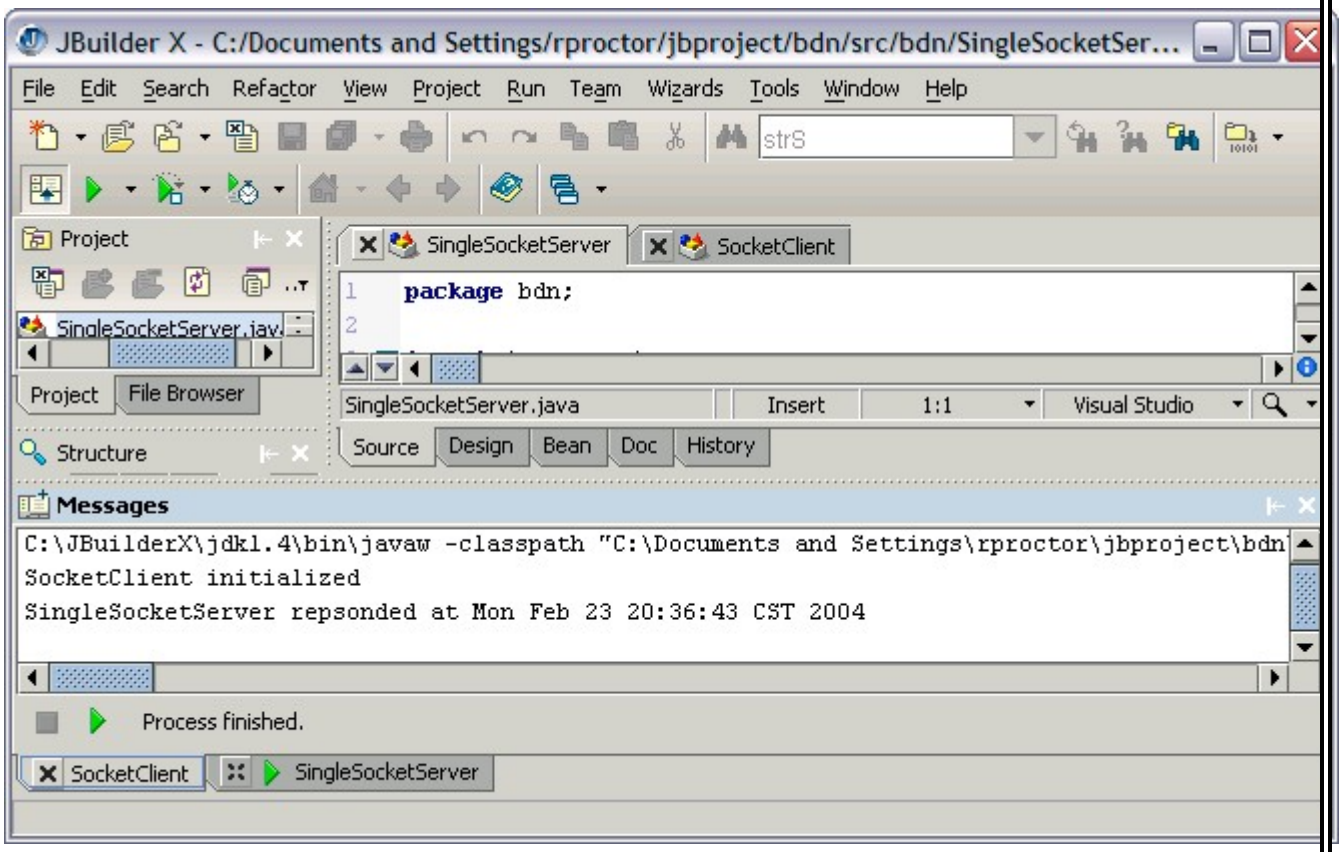


Figure 3: SocketClient After Connecting with SingleSocketServer

Notice the results. Figure 2 shows messages for the SingleSocketServer class. From the message, we can see that the SockeClient called the server at 20:36:33. Looking at Figure 3 we see the messages for the SocketClient class. The message tells us that the server responded at 20:36:43...10 seconds later. SocketClient has ended and SocketServer is waiting for another connection.

Let's take the same scenario and add a second instance of SocketClient to the mix (to do this, start SocketClient, and while it's running start a second instance).

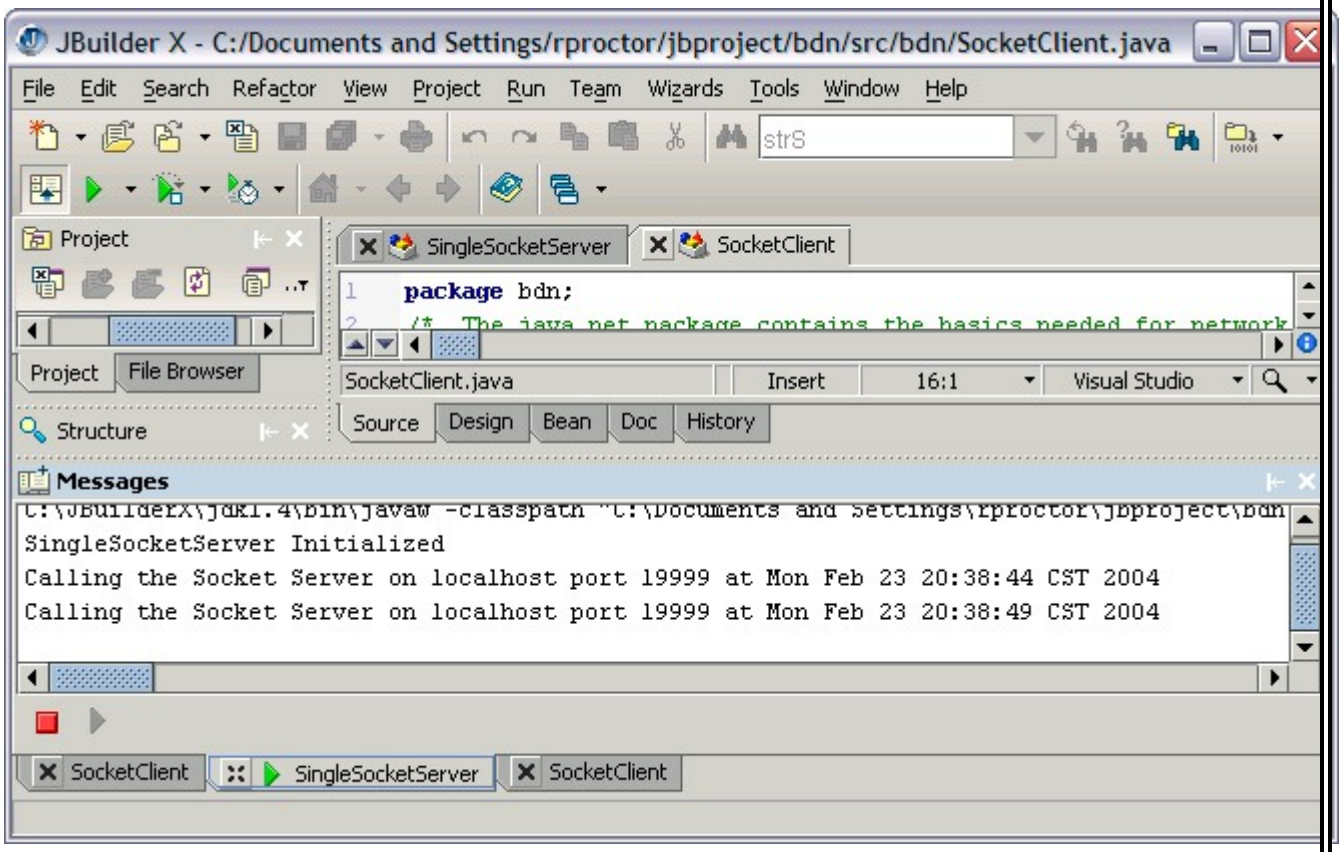


Figure 4: SingleSocketServer with Two SocketClient Instances

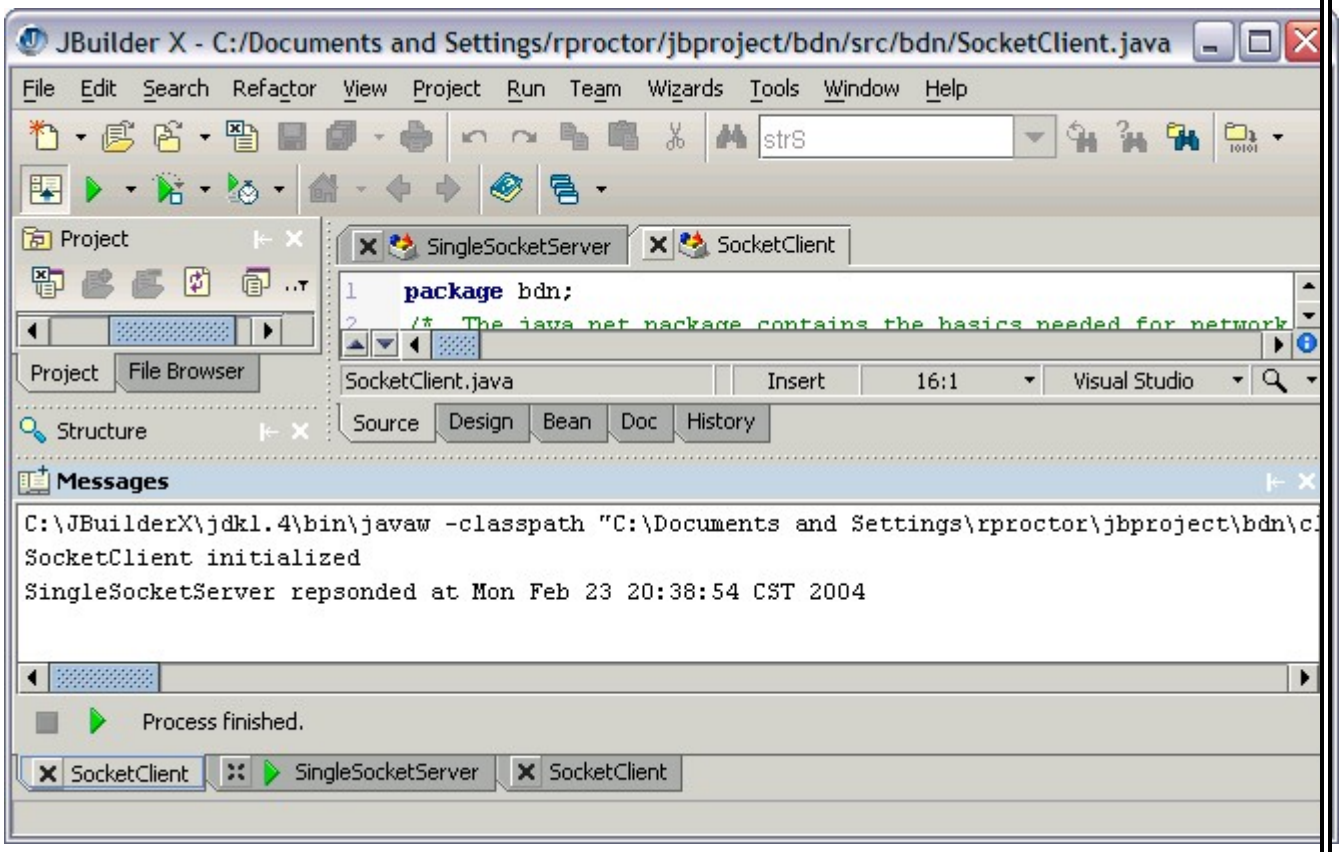


Figure 5: First Instance of SocketClient

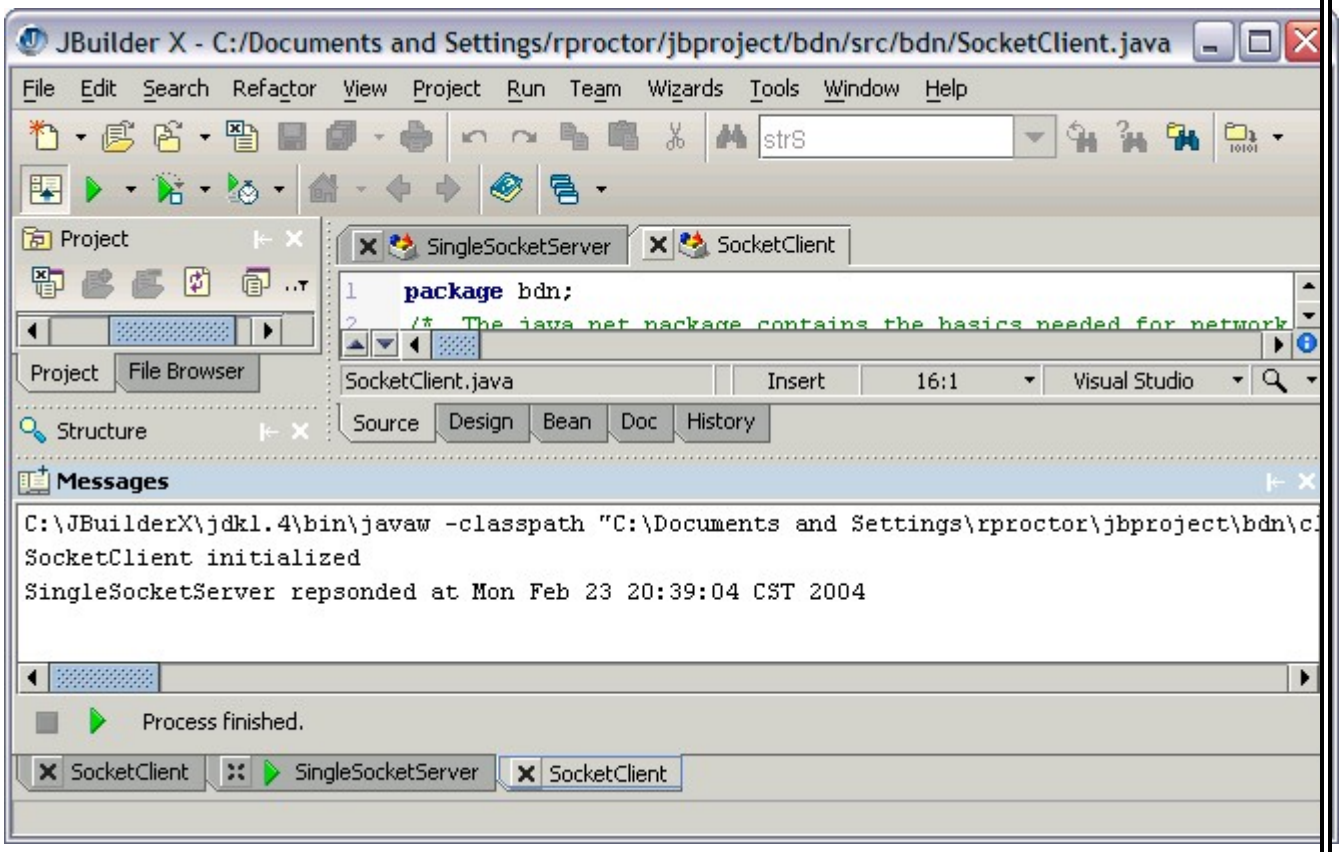


Figure 6: Second Instance of SocketClient

This time look at the results. Figure 4 shows that SingleSocketServer logs two messages, showing the TimeStamp that each instance of SocketClient. Figure 5 shows the first instance of SocketClient we executed. Notice that the time in Figure 5 (20:38:54) is 10 seconds later than the time in the first message in Figure 4 (20:38:44)...as expected

Now let's look at the second instance of SocketClient. From the time in Figure 4 we see that the second instance was launched 5 seconds after the first one. However, looking at the time in Figure 6, we can see that SingleSocketServer didn't respond to the second instance until 10 seconds after it responded to the first instance of SocketClient…13 seconds later. The reason for this is simple. SingleSocketServer is running in a single thread and each incoming socket connection must be completed before the next one can start.

Now, you might be asking, "Why in the world would someone want process only one socket at a time?" Recently I was working on a project where a non-java program had to be launched across a network from a java program. And, only one instance of the non-java program could be running at a time. I used a similar solution to solve that problem. Also, it can be a handy way of keeping multiple instances of the same java program from being launched at the same time. "That's fine" you say, "but I want my server to accept multiple sockets. How do you do that?"

MultipleSocketServer: A Server That Handles Multiple Client Connections

```
package bdn;
import java.net.*;
import java.io.*;
import java.util.*;

public class MultipleSocketServer implements Runnable {

  private Socket connection;
  private String TimeStamp;
  private int ID;
  public static void main(String[] args) {
  int port = 19999;
  int count = 0;
    try{
      ServerSocket socket1 = new ServerSocket(port);
      System.out.println("MultipleSocketServer Initialized");
      while (true) {
        Socket connection = socket1.accept();
        Runnable runnable = new MultipleSocketServer(connection, ++count);
        Thread thread = new Thread(runnable);
        thread.start();
      }
    }
    catch (Exception e) {}
  }
MultipleSocketServer(Socket s, int i) {
  this.connection = s;
  this.ID = i;
}
```

We're not going to spend much time examining all the code here since we've already been through most of it. But, I do want to highlight some things for you. Let's start with the class statement. We're introducing a new concept here. This time we're implementing the Runnable interface. Without diving deep into the details of threads, suffice it to say that a thread represents a single execution of a sequential block of code.

In our previous example, the SingleSocketServer class had a while(true) block that started out by accepting an incoming socket connection. After a connection was received, another connection could not happen until the code looped back to the connection = socket1.accept(); statement. This block code represents a single thread…or at least part of a single thread of code. If we want to take this block of code and make it into many threads…allowing for multiple socket connections...we have a couple of options: extending the implementing the Runnable interface or extending the Thread class. How you decide which one to use is entirely up to your needs. The Thread class has a lot of methods to do various things like control thread behavior. The Runnable interface has a single method run() (Thread has this method too). In this example, we're only concerned with the run() method…we'll stick with the Runnable interface.

Let's continue…

Looking through the main() method, we still have to set up our server to allow for connections on Port 19999, there's still a while(true) block, and we're still accepting connections. Now comes the difference. After the connection is made, we instantiate a Runnable object **runnable** using a constructor for MultipleSocketServer that has 2 arguments: a Socket object **connection** for the socket that we've just accepted, and an int **count**, representing the count of open sockets. The concept here is simple: we create a new socket object for each socket connection, and we keep a count of the number of open connections. Although we're not going to worry about the number of connections in this example, you could easily use count to limit the number of sockets that could be open at once.

Once we've instantiated **runnable**, we instantiate a new Thread **thread** by passing **runnable** to the Thread class. We call the start() method of thread and we're ready go. Invoking the start() method spawns a new thread and invokes the object's run() method. The actual work within the thread happens within the run() method. Let's take a quick look at that now.

The run() Method

```
public void run() {
   try {
     BufferedInputStream is = new BufferedInputStream(connection.getInputStream());
     InputStreamReader isr = new InputStreamReader(is);
     int character;
     StringBuffer process = new StringBuffer();
     while((character = isr.read()) != 13) {
       process.append((char)character);
     }
     System.out.println(process);
     //need to wait 10 seconds to pretend that we're processing something
     try {
       Thread.sleep(10000);
```

```
      }
      catch (Exception e){}
      TimeStamp = new java.util.Date().toString();
      String returnCode = "MultipleSocketServer repsonded at "+ TimeStamp + (char) 13;
                                          BufferedOutputStream os = new
BufferedOutputStream(connection.getOutputStream());
      OutputStreamWriter osw = new OutputStreamWriter(os, "US-ASCII");
      osw.write(returnCode);
      osw.flush();
    }
    catch (Exception e) {
      System.out.println(e);
    }
    finally {
      try {
        connection.close();
      }
      catch (IOException e){}
    }
}
}
```

The run() method's responsibility is to run the block of code that we want threaded. In our example the run() method executes the same block of code we built in the SingleSocketServer class. Once again, we read the BufferedInputStream, pretend to process the information, and write a BufferedOutputStream. At the end of the run() method, the thread dies.

Now it's time to run our MultipleSocketServer. Using the same scenario we did previously. Let's look at what happens when we run the MultipleSocketServer and two instances of SocketClient.

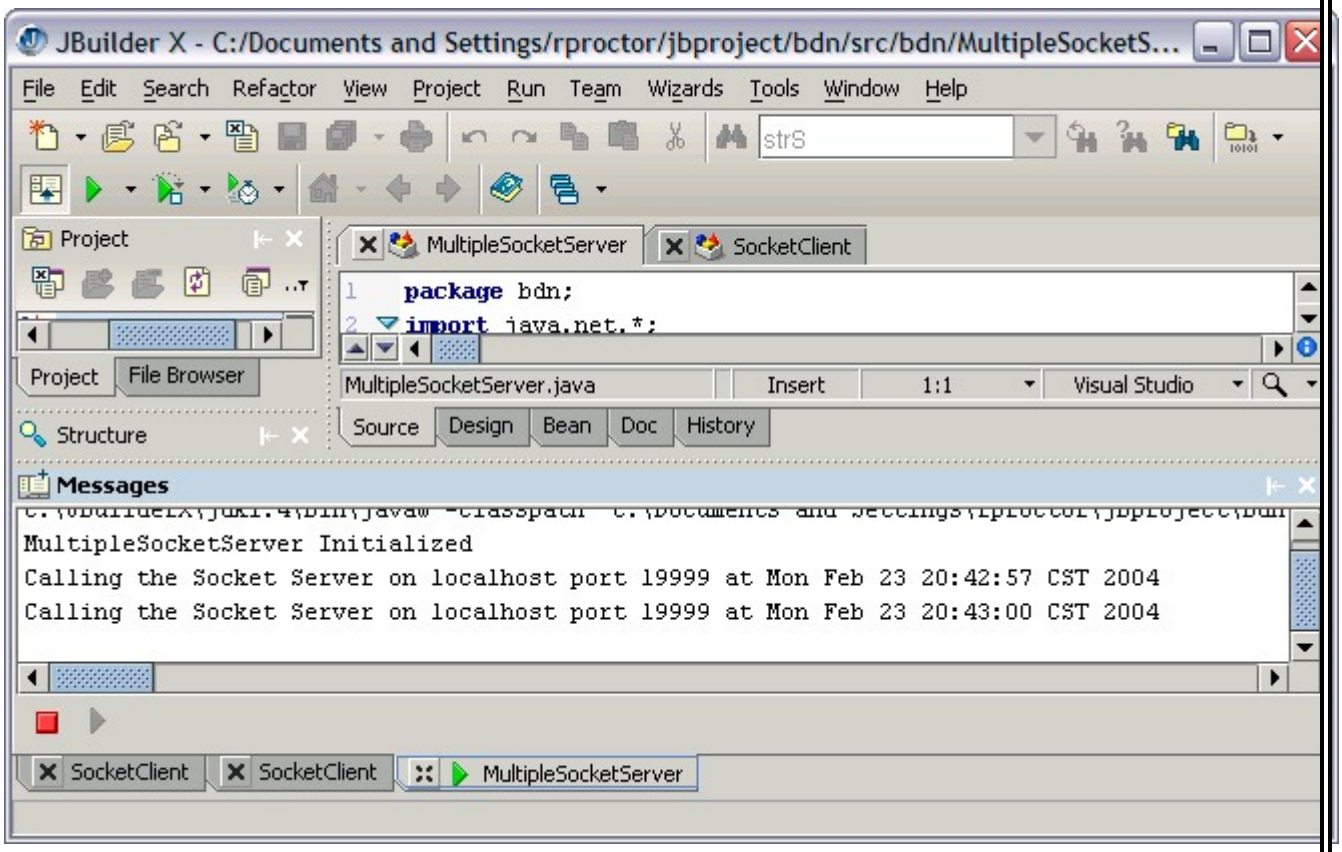


Figure 7: MultipleSocketServer with Two Instances of SocketClient

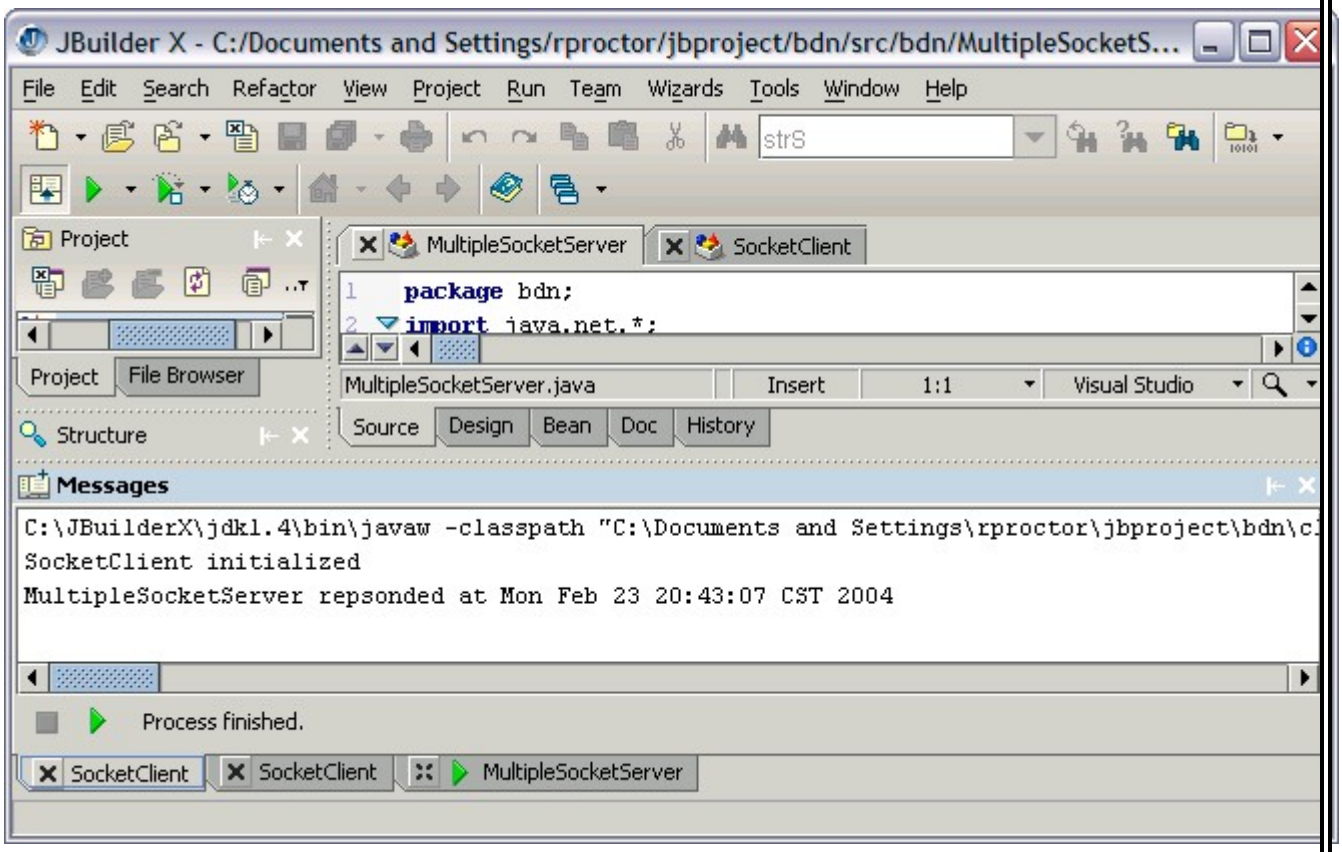


Figure 8: First Instance of SocketClient

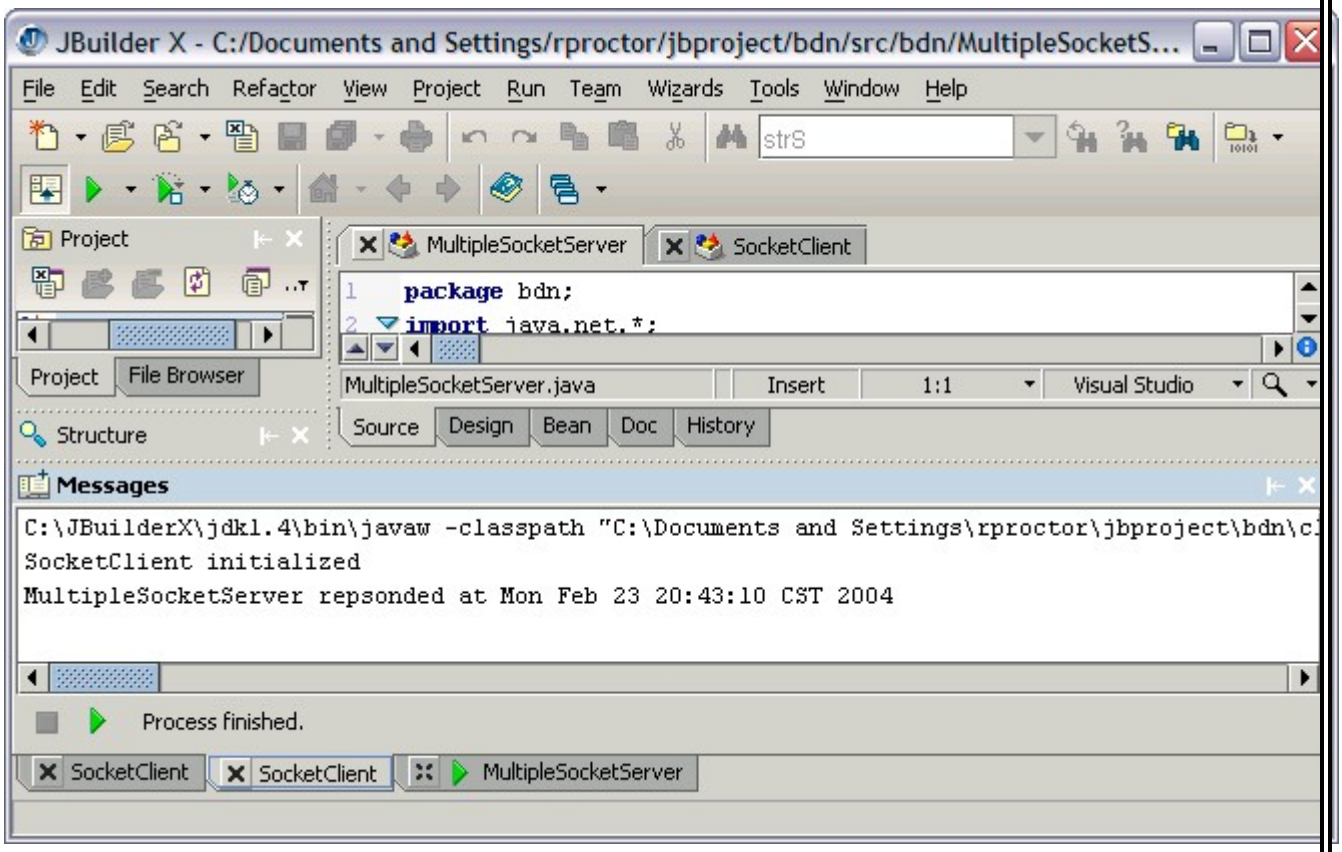


Figure 9: Second Instance of SocketClient

Looking at the messages in Figure 7 that come from the MultipleSocketServer class, we can see that requests for socket connections were sent by the SocketClient programs within a few seconds of each other. According to the console messages on the first instance of SocketClient in Figure 8, the server responded 10 seconds after the request was sent. Figure 9 shows that the second instance of SocketClient receives a response 10 seconds after it sent the request also, thus allowing us the capability of processing multiple socket connections simultaneously.

Summary

Network programming in Java revolves around sockets. Sockets allow us to communicate between programs across the network. Java's approach to socket programming allows us to treat socket I/O the same as we do any other I/O…utilizing InputStreams and OutputStreams. Although, the examples presented here are relatively simple, they give you an idea of the power of Java in the Client-Server world.

Next time, we'll move a level up from socket programming and deal with the basics of calling objects across a network using Java's Remote Method Invocation (RMI) facility.

Client/Server Programming

The client/server model is an application development architecture designed to separate the presentation of data from its internal processing and storage. This paradigm is based on the theory that the rules which operate on data do not change no matter how many applications are accessing that data. An airline might want to allow customers to purchase tickets on the Web, through a travel agent, or even through ATM-like machines located in airports and malls. No matter which interface the customer uses, the number of seats available does not change, nor does the fact that you cannot sell a 21-day advance fare ten days before the flight. The client/server model would enable the airline to build the application so that each interface accesses the same seat availability data and calls the exact same rules to validate the sale of a ticket.

With the rapid rise of the World Wide Web, the power of the client/server paradigm now serves the masses. In feeding the same exact data to multiple client browsers across the Internet, even the most basic HTML page fits the client/server model in its simplest form. Tools like CGI, and now Java, enhance the Web's use of the client/server model by consolidating application rules on the Web server.

Java's Suitability for Client/Server Programming

The Web provides an excellent example of the most basic reason for separating the presentation of data from its storage and processing. As a Web developer, you have absolutely no control over the platforms and software with which users are accessing your data. You might consider writing your application for each potential platform that you are targeting. For the airline ticketing system discussed in the previous example, this approach forces you to recode the rule for a 21-day advance fare three times if that rule changes. Obviously, writing the application for each possible platform is a recipe for a maintenance nightmare.

In addition to having little control over the systems being used for the presentation of your data, a complex application often has different needs, which are often best met by different hardware or operating systems. An ATM-like machine in a mall designed specifically for selling airline tickets has no need for the hardware required by home computers to provide a graphical user interface. Similarly, the user's home computer is generally not well-suited for acting as a massive data storage device on the level required by such an application.

The primary selling point of Java as a programming language, the Java virtual machine, also provides its primary selling point as a tool in client/server development. With code portability difficult to deliver in any other programming language, Java instead enables developers to write the user interface code once, distribute it to the client machine, and have the client machine interpret that presentation in a manner that makes sense for that system.

Beyond architecture independence, Java provides a rich library of network enabled classes that allow applications ready access to network resources in the form of traditional TCP/IP addressing and URL referencing. New tools, such as JavaSoft's Remote Method Invocation (RMI), promise only to extend Java's network usability.

The final beauty of Java in client/server development involves deployment strategies. In traditional client/server systems, deployment of an application requires users to physically install the client portion of the application on their machine. A Java client system, on the other hand, can be executed from across the network. As a result, client machines are always running the most current version of the application.

Client and Servers

Client systems generally have a clear separation from the servers with which they work. The underlying mechanics of the system are hidden from users who generally only need a portion of the functionality provided by the server system. The client application serves a particular problem domain, such as order entry, accounting, game playing, or ticket purchasing, and talks to the server through a narrow, well-defined network interface.

The server portion of a client/server application manages resources shared among multiple users, often accessing the server through multiple client front-ends. A Web server, for example, delivers the same HTML pages across the Internet to Web users. More complex applications, such as business database applications, enable clients to make query requests through the server and receive the results.

Merging the Client and Server

Developers commonly use one of two client/server architectures in system design:

- Two-tier architecture
- Three-tier architecture

The simple retrieval and display of information part of serving HTML pages is an example of a two-tier client/server. On one end, or tier, data is stored and served to clients. On the other end, that data is displayed in a format that fits the situation.

On many systems, however, the retrieval and display of data forms only a fraction of the system. A complex business system generally involves the processing of data before it can be displayed or saved back into storage. In a two-tier system, the client handles a

majority of this extra processing. This heavily loaded client is often referred to as a "fat client."

A two-tier design using a fat client provides a quick and dirty architecture for building small, non-critical systems. The fat client architecture shows its dirty side in maintenance and scalability. With data processing so tightly coupled to the GUI presentation, user interface changes necessitate working around the more complex business rules. In addition, the two-tier system ties the client and server together so tightly that distributing the data across databases becomes difficult.

Three-tier client/server design mandates that data processing should be separated from the user interface and data storage layers. Stored procedures provide the most common method of intervening between user interface and data storage in a third tier. A stored procedure performs complex database queries and updates, returning the results to the client.

While two-tier development simply separates data storage from presentation, the three-tier system consists of the following layers:

- user interface
- data processing, or business rules
- data storage

In isolating application functionality in three-tier development, the system becomes easier to maintain and modify. The user interface, for example, no longer cares where or how the system stores its data. Changes in data storage, such as distributing the data across multiple databases, ends up having a much smaller impact on the system as a whole.

Java's Deployable Code Advantage

The Web provides developers with an ideal application deployment infrastructure, especially when Java is part of the picture. Unlike other client/server development tools, Java is distributed at runtime across the Internet to client machines. By storing the application on a central server and downloading it at runtime, the user is always using the latest release of the software. Consider the following URL:

appletviewer http://www.strongsoft.com/Java/test.html

Java TCP/IP Sockets

TCP/IP sockets form the basic mode of data communication on the Internet. The Java Development Kit addresses TCP/IP programming requirements through a high level suite of APIs and TCP/IP streams. An application can use these streams to enable network input and output to be manipulated in a variety of ways. The java.io package from the standard Java release defines these forms, which include DataInputStreams, DataOutputStreams, and PrintStreams.

The package java.net has the backbone TCP/IP classes provided by Java. From these classes, the application can create the data I/O streams from java.io that it needs for network communication. Java provides these network classes in java.net for its socket support:

- DatagramSocket
- Socket
- SocketImplFactory
- SocketImpl
- ServerSocket

Platform-specific implementations extend the abstract class SocketImpl to perform the low-level network interfacing, which varies from system to system. The high level Socket and ServerSocket classes in turn reference the system's particular SocketImpl class for network access. Because the default implementations of the basic JDK classes are not firewall-aware, it is possible to extend the SocketImplFactory and SocketImpl classes to provide firewall functionality.

Using Datagram for UDP Sockets

Starting with a ticker tape applet, TickerTape, the following listing features a facility for subscribing to a broadcast ticker tape server. The server is designed to broadcast messages to a list of connected clients. Because the reception of particular messages in a ticker tape system is unimportant, this example uses the simplest form of sockets, the DatagramSocket. A datagram uses UDP, or Unreliable Datagram Protocol. UDP is a broadcast protocol that does not guarantee the reception of its messages. Because they do no reception checking, however, UDPs have a lower resource overhead than protocols which perform error checking. If you are sending out roughly the same information repeatedly (as is being done in Listing 34.1) then the resource savings outweigh any problems related to losing a packet every now and then.

Listing 34.1. The client applet, TickerTape.java.

```
import java.awt.Color;
import java.awt.Graphics;
import java.net.*;
import java.io.InputStream;
import java.util.Date;

/** */

public class TickerTape extends java.applet.Applet implements Runnable {
   String message_error = "Error - No Message…";
   String message;

   /** The width of the string in pixels.  No need to know this since
      we can reset the string */
   int messageWidth;

   /** keep track of where we are printing the current string */
   int position;

   /** Thread */
   Thread ticker = null;

   /** Just a way to check if the thread is suspended or not.  If
      through some bug this gets set wrong it just prints
      wrong commands */
   boolean suspend = false;

   /* The amount of time to rest between redrawing line */
   int rest = 100;

   /** amount of space to jump.  Hopefully negative numbers will
      move it in the other direction */
   int jump = 5;

   public void init() {
      String tmpParam;

      tmpParam = getParameter("jump");
      if( tmpParam != null ) {
         jump = new Integer(tmpParam).intValue();
         if( jump == 0 ) {
            jump = 5;
            System.out.println("Zero value for jump: using 5");
         }
      }
```

```
    tmpParam = getParameter("rest");
    if( tmpParam != null ) {
       rest = new Integer(tmpParam).intValue();
       if( rest < 0 ) {
          rest = 100;
          System.out.println("Negative rest value: using 100");
       }
    }
    message = getMessage();
    if( message == null ) message = message_error;

    messageWidth = getFontMetrics(getFont()).stringWidth(message);
    position = (jump < 0) ? -messageWidth : size().width;
    setForeground(Color.red);
    setBackground(Color.white);
}

public void start() {
    if( ticker == null ) {
       ticker = new Thread(this);
       ticker.start();
    }
}

public void stop() {
    ticker = null;
}

public void run() {
    while( ticker != null ) {
       repaint();
       if( ((jump < 0) && (position > size().width)) ||
         (position < -messageWidth) )
           position = (jump < 0) ? -messageWidth : size().width;
       try Thread.sleep(rest);
       catch( InterruptedException e ) ticker = null;
       position -= jump;
    }
}

public void paint(Graphics g) {
    g.drawString(message, position, getFont().getSize());
}

public String getMessage() {
    int port;
```

```
        InetAddress address;
        DatagramSocket socket;
        DatagramPacket packet;
        byte[] sendBuf = new byte[256];

        try {
           socket = new DatagramSocket();
           port = 1111;
           address = InetAddress.getByName("StrongSun");
           packet = new DatagramPacket(sendBuf, 256, address, port);
           socket.send(packet);
           packet = new DatagramPacket(sendBuf, 256);
           socket.receive(packet);
           message = new String(packet.getData(), 0);
           socket.close();
        }
        catch( Exception e ) {
           System.err.println("Exception: " + e);
           e.printStackTrace();
        }
        return message;
     }
}
```

The HTML code for this applet's Web page follows:

```
<TITLE>Ticker Tape Applet Using UDP</TITLE>
<H1>Test of the TickerTape Datagram Applet</H1>

<HR>

<APPLET CODE="TickerTape" WIDTH=400 HEIGHT=25>
<PARAM NAME=jump VALUE="7">
TickerTape applet not loaded!
</APPLET>
<HR>
```

The TickerTape applet uses a loop in a second thread which repeatedly calls the getMessage() method. This method returns a String to use as the scrolling text from the ticker tape server. It does this first by instantiating a new DatagramSocket and sending a request to the server. The server responds with a String to be painted for the user. Listing 34.2 provides the server code.

Listing 34.2. The TickerTapeServer application, TickerTapeServer.java.

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

class TickerTapeServer extends Thread{
   private DatagramSocket socket = null;
   private String broadcastMessage = "TickerTapeServer Messages Here!";

   TickerTapeServer() {
      super("TickerTapeServer");
      try {
         socket = new DatagramSocket(1111);
         System.out.println("TickerTapeServer listening on port: " +
                  Âsocket.getLocalPort());
      } catch (java.net.SocketException e) {
         System.err.println("Could not create datagram socket.");
      }
   }

   public static void main(String[] args) {
      if (args.length != 1) {
         System.out.println("Usage: java TickerTapeServer " +
           "'your text here in quotes' ");
         return;
      }
      TickerTapeServer tts = new TickerTapeServer();
      tts.start();
      tts.setBroadcastMessage (args[0]);
   }

   public void run() {
      if (socket == null) return;
      while (true) {
         try {
            byte[] buf = new byte[256];
            DatagramPacket packet;
            InetAddress address;
            int port;
            String dString = null;

            packet = new DatagramPacket(buf, 256);
            socket.receive(packet);
            address = packet.getAddress();
            port = packet.getPort();
            dString = getBroadcastMessage();
            dString.getBytes(0, dString.length(), buf, 0);
```

```
            packet = new DatagramPacket(buf, buf.length, address, port);
            socket.send(packet);
          } catch (Exception e) {
            System.err.println("Exception:  " + e);
            e.printStackTrace();
          }
        }
    }

    public String getBroadcastMessage () {
      return broadcastMessage;
    }

    public void setBroadcastMessage (String s) {
      broadcastMessage = s;
    }
}
```

Naturally, before any client can connect, you need to start the server application. To start it, you simply issue the following command:

java TickerTapeServer 'My broadcast message'

Client applications see whatever message you specify on the command line. In a second thread, the server application waits for clients to connect before sending them the broadcast message. Once it receives the client request, the server application grabs the client's address and sends the broadcast message back to the client.

Using Socket and ServerSocket for TCP Sockets

TCP is a more reliable form of communication than UDP. Unlike UDP, TCP sockets perform error-checking to ensure the packets are delivered to their destination. TCP sockets are connection-based sockets, meaning that a TCP socket is a two-way form of communication maintained until one side or the other breaks it off. This contrasts with the connectionless broadcast essence of UDP.

In order to create a TCP-based client/server example, you first need to build a framework for network access.

A Simple Connection

Creating a TCP connection to a server involves only the following code fragment:

java.net.Socket connection;

```
try {
   connection = new java.net.Socket("athens.imaginary.com", 1701)
} catch( Exception e ) {
}
```

The constructor for the Socket class requires a host with which to connect, in this case "athens.imaginary.com", and a port number, which is the port of a mud server. If the server is up and running, the code creates a new Socket instance and continues running. If the code encounters a problem with connecting, it catches the problem in the form of an exception. To disconnect from the server, the application should call:

```
connection.disconnect();
```

A simple socket client looks like the following:

```
public class BasicClient {
   boolean active;
   java.net.Socket connection;

   public BasicClient(String address, int port) {
      try {
         connection = new java.net.Socket(address, port);
         active = true;
      }
      catch( java.io.IOException e ) {
         active = false;
      }
   }

   public void done() {
      if( !active ) return;
      connection.close();
      active = false;
   }
}
```

Socket I/O is blocking in nature, meaning that when an application tries to read from a socket, all processing in that thread comes to a halt until something is read from that socket. Fortunately, Java is very friendly to multithreaded programming. Socket programmers can use Java threads to read from a socket in one thread and write to it in another, and perhaps perform additional processing in another. This extended version of our basic client implements it with a multi-threaded structure:

```
public class BasicClient implements Runnable{
   private boolean active = false;
   private java.net.Socket connection = null;
```

```
private Thread thread = null;
private String address;
private int port;

public BasicClient(String addr, int p) {
   address = addr;
   port = p;
}

public void start() {
   if( thread == null ) {
      thread = new Thread(this);
      thread.start();
   }
}

public void stop() {
   if( thread != null ) {
      thread.stop();
      thread = null;
   }
   if( active ) {
      if( connection != null ) {
         try {
            connection.close();
            active = false;
            connection = null;
         }
         catch( java.io.IOException e ) {
         }
      }
      else active = false;
   }
}

public void run() {
   try {
      connection = new java.net.Socket(address, port);
      active = true;
   }
   catch( java.io.IOException e ) {
      active = false;
      System.out.println("Failed to connect to server.");
   }
}
```

```
    public boolean isActive() {
       return active;
    }
}
```

A Music Store for the Web

Retail applications are simple client/server uses of the Internet that provide a perfect example of how to structure such a program in Java. Any retail application first requires a server program that provides data to customers and takes their orders; then it needs a client program that provides the interface that allows them to view a product line and enter purchase requests.

Of course, any system involving the exchange of money over the Internet has some hefty security requirements. For the sake of simplicity, however, we will ignore security requirements and deal with the basic building blocks of socket-based client/server programming. Our application, a music store for the Web, should therefore have the following functionality:

1. Start the server with the name of a JDBC driver and the JDBC URL for accessing it.
2. Wait for connections from client systems.
3. For each client request, provide a list of available titles and wait for purchase requests.
4. Enter any purchase requests into the database to be processed later.
5. Shut down the server.

Though most of the time it is simply listening for incoming client connections, the server is responsible for quite a bit of work. Listing 34.3 provides the server portion of the application.

Listing 34.3. The Web music store server.

```
import java.net.Socket;
import java.net.ServerSocket;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class Server extends Thread {
  private Connection connection;
  private Socket socket;

  public Server(Socket sock, Connection conn) {
    socket = sock;
    connection = conn;
  }
```

```
public void run() {
  java.io.DataInputStream input;
  java.io.PrintStream output;

  try {
    String tmp;
    java.util.StringTokenizer tokens;
    java.util.Vector albums = new java.util.Vector();
    boolean transacting = true;
    Statement statement;
    ResultSet result_set;

    input = new java.io.DataInputStream(socket.getInputStream());
    output = new java.io.PrintStream(socket.getOutputStream());
    statement = connection.createStatement();
    result_set = statement.executeQuery("SELECT album_id, artist, title, "+
                                 &quo t;price " +
                                & nbsp;"FROM album ");
    while( result_set.next() ) {
      String id, artist, title, price;

      id = result_set.getString(1);
      artist = result_set.getString(2);
      title = result_set.getString(3);
      price = result_set.getString(4);
      albums.addElement(id + ":" + artist + ":" + title + ":" + price);
    }
    output.println(albums.size());
    for(int i = 0; i<albums.size(); i++)
      output.println((String)albums.elementAt(i));
    while( transacting ) {
      tmp = input.readLine();
      tokens = new java.util.StringTokenizer(tmp);
      tmp = tokens.nextToken();

      if( tmp.equals("exit") ) {
        transacting = false;
      }
      else if( tmp.equals("purchase") ) {
        String credit_card;
        String id;

        if( tokens.countTokens() != 2 ) {
          output.println("error Invalid command");
          socket.close();
```

```
            return;
          }
          id = tokens.nextToken();
          credit_card = tokens.nextToken();
          statement = connection.createStatement();
          statement.executeUpdate("INSERT INTO purchase (" +
                                  "credit_card, album) " +
                                  "VALUES('" + credit_card + "', '" +
                                  id + "')");
          output.println("ok");
        }
      }
    }
    catch( Exception e );
    finally {
      try {
        socket.close();
      }
      catch( java.io.IOException e );
    }
  }

  static public void main(String args[]) {
    ServerSocket port_socket;
    String driver;
    String url;
    int port;

    if( args.length != 3 ) {
      System.err.println("Syntax: java Server <JDBC driver> <JDBC URL> " +
                         "<port>");
      System.exit(-1);
      return;
    }
    driver = args[0];
    url = args;
    try {
      port = Integer.parseInt(args);
    }
    catch( NumberFormatException e ) {
      System.err.println("Invalid port number: " + args);
      System.exit(-1);
      return;
    }
    try {
      port_socket = new ServerSocket(port);
```

```
    }
    catch( java.io.IOException e ) {
      System.err.println("Failed to listen to port: " + e.getMessage());
      System.exit(-1);
      return;
    }
    while( true ) {
      try {
        Connection conn;
        Server server;
        Socket client_sock = port_socket.accept();

        conn = java.sql.DriverManager.getConnection(url, "user", "pass");
        server = new Server(client_sock, conn);
        server.start();
      }
      catch( java.io.IOException e ) {
        System.err.println("Connection failed.");
      }
      catch( java.sql.SQLException e ) {
        System.err.println("Failed to connect to database.");
      }
    }
  }
}
```

The application uses the main thread of the server simply to listen to the network for connections. Each time it accepts a connection, it creates a new instance of itself to handle the client/server communication in a separate thread.

The most tedious and most difficult aspect of client/server programming with sockets involves the actual protocol you create for the communication. The music store server uses a very simple protocol for communicating with a client. It simply sends it a full list of all titles in stock, then waits for either a purchase request or an end processing notification. Even with this simplistic protocol, however, we have to handle the parsing of each string sent by the client.

The core Java libraries do help simplify protocol management through the StringTokenizer utility. This class breaks up a string into individual tokens based on a delimiter. By default, the delimiter is a space. With purchase requests, we expect a string in the form "purchase <album id> <credit card number>". The first token thus is the purchase command, the second token the album ID, and the third token the credit card number used to purchase the album.

Listing 34.4 provides the socket code for the client end of the application. It assumes that some sort of user interface is built on top of it.

Listing 34.4. The music store client socket code.

```
import java.io.DataInputStream;
import java.io.PrintStream;
import java.net.Socket;

public class Client {
  private Socket socket;
  private String host;
  private int port;
  private java.util.Vector albums = new java.util.Vector();
  private DataInputStream input;
  private PrintStream output;

  public Client(String h, int p) throws java.io.IOException {
    String data[] = new String;
    java.util.StringTokenizer tokens;
    String tmp;
    int x;

    host = h;
    port = p;
    socket = new Socket(host, port);
    input = new DataInputStream(socket.getInputStream());
    output = new PrintStream(socket.getOutputStream());
    tmp = input.readLine();
    try {
      x = Integer.parseInt(tmp);
    }
    catch( NumberFormatException e ) {
      throw new java.io.IOException("Communication error, invalid " +
                                    "number of albums.");
    }
    while( x-- > 0 ) {
      tmp = input.readLine();
      tokens = new java.util.StringTokenizer(tmp, ":");
      if( tokens.countTokens() != 4 ) {
        throw new java.io.IOException("Invalid album format.");
      }
      for(int i=1; i<=4; i++) data[i] = tokens.nextToken();
      albums.addElement(data);
    }
  }
```

```
  public synchronized void close() throws java.io.IOException {
    output.println("exit");
    socket.close();
  }

  public String[][] getAlbums() {
    String album_data[][];

    synchronized(albums) {
      album_data = new String[albums.size()][];
      albums.copyInto(album_data);
      return album_data;
    }
  }

  public synchronized void purchaseAlbum(String id, String cc)
  throws java.io.IOException {
    String tmp;

    output.println("purchase " + id + " " + cc.trim());
    tmp = input.readLine();
    if( tmp.equals("ok") ) return;
    else throw new java.io.IOException(tmp);
  }
}
```

Again, protocol negotiation differentiates this code from the basic client code shown earlier in the chapter. When the applet or application creates this Client object, it connects to the Server program and gets a list of all albums. Upon receiving an album, it uses the StringTokenizer in a slightly different fashion to break up the string into its components. As you saw in the Server code, information about an album is packed into a single string separated by a ":". By default, the StringTokenizer splits the string on a space. To change the delimiter, it needs a second argument to its constructor, the string to serve as the delimiter. In this case, we passed a ":".

The rest of this client code simply provides methods for the user interface to communicate with the server. Specifically, it allows the user interface to close the connection, get the list of albums, and purchase an album.

Summary

The basic building blocks for client/server programming are the IP sockets that form the communication layer of the Internet. While socket programming can be very tedious and time consuming, Java has provided classes designed to minimize this tedium to enable

developers to harness the power of client/server programming. The DatagramSocket, ServerSocket, and Socket classes all provide access to the IP protocols themselves. The DataInputStream, DataOutputStream, and PrintStream classes provide access to the data. Finally, the StringTokenizer class provides simple data manipulation.

The most important factor in creating a socket communication layer is understanding exactly what your application should communicate. You cannot create the necessary communication protocol if you do not fully understand what the client and server need to be saying to each other.

An **e-mail client** (also **mail user agent** (MUA) or **e-mail reader**) is a frontend computer program used to manage e-mail.

Sometimes, the term e-mail client is also used to refer to any agent acting as a client toward an e-mail server, independently of it being a real MUA, a relaying server, or a human typing directly on a telnet terminal. In addition, a web application providing the relevant functionality is sometimes considered an e-mail client.

Functionality and configuration

Although mail user agents aim at enabling users to deal with their mail with minimal technical knowledge, some understanding of the operations involved is useful for making configuration decisions appropriate to the user's requirements.

Retrieving messages from a mailbox

Like most client programs, an MUA is only active when a user runs it. Messages arrive on the Mail Transfer Agent (MTA) server. Unless the MUA has access to the server's disk, messages are stored on a remote server and the MUA has to request them on behalf of the user.

In the first case, shared disk, a user logs on a server and runs an MUA on that machine. The MUA reads messages from a conventionally formatted storage, typically mbox, within the user's HOME directory. The MTA uses a suitable mail delivery agent (MDA) to add messages to that storage, possibly in concurrence with the MUA. This is the default setting on many Unix systems. Webmail applications running on the relevant server can also benefit from direct disk access to the mail storage.

For personal computing, and whenever messages are stored on a remote system, a mail user agent connects to a remote mailbox to retrieve messages. Access to remote mailboxes comes in two flavors. On the one hand, the Post Office Protocol (POP) allows the client to download messages one at a time and only delete them from the server after they have been successfully saved on local storage. It is possible to leave messages on the server in order to let another client download them. However, there is no provision for flagging a specific message as seen, answered, or forwarded, thus POP is not convenient for users who access the same mail from different machines or clients. On the other hand,

the Internet Message Access Protocol (IMAP) allows users to keep messages on the server, flagging them as appropriate. IMAP provides sub-folders. Typically, the Sent, Drafts, and Trash folders are created by default.

Both POP and IMAP clients can be configured to access more mailboxes at the same time, as well as to check each mailbox every given number of minutes. IMAP features an idle extension for real time updates, providing faster notification than polling where long lasting connections are feasible.

Client settings require the server's name or IP address, and the user name and password for each remote incoming mailbox.

Formatting messages

Mail user agents usually have built-in the ability to display and edit text. Editing HTML text is a popular feature. Invoking an external editor may be an alternative.

MUAs responsibilities include proper formatting according to RFC 5322 for headers and body, and MIME for non-textual content and attachments. Headers include the destination fields, To, Cc, and Bcc, and the originator fields From which is the message's author(s), Sender in case there are more authors, and Reply-To in case responses should be addressed to a different mailbox. To better assist the user with destination fields, many clients maintain one or more address books and/or are able to connect to an LDAP directory server. For originator fields, clients may support different identities.

Client settings require the user's real name and e-mail address for each user's identity, and possibly a list of LDAP servers.

Submitting messages to a server

As a basic function, an MUA is able to introduce new messages in the transport system. Typically, it does so by connecting to either an MSA or an MTA, two variations of the SMTP protocol. The client needs to put a message quickly without worrying about where the message eventually will be delivered: that's why a transport system exists. Thus it always connects to the same preferred server, however, how does that server know that it should accept and relay submissions from that client? There are two ways. The older method recognizes the client's IP address, e.g. because the client is on the same machine and uses internal address 127.0.0.1, or because the client's IP address is controlled by the same internet service provider that provides both internet access and mail services. The newer method, since the SMTP protocol has an authentication extension, is to authenticate. The latter method eases modularity and nomadic computing.

Client settings require the name or IP address of the preferred outgoing mail server, the port number (25 for MTA, 587 for MSA), and the user name and password for the authentication, if any. There is a non-standard port 465 for SSL encrypted SMTP sessions, that many clients and servers support for backward compatibility. Transport

Layer Security encryption can be configured for the standard ports, if both the client and the server support it.

Encryption

With no encryption, much like for postcards, e-mail activity is plainly visible by any occasional eavesdropper. E-mail encryption enables to safeguard privacy by encrypting the mail sessions, the body of the message, or both. Without it, anyone (examples: the government (warrantless wiretapping, great firewall of China), fellow wireless network users such as at an Internet cafe or other public network, whether the network is open or not) with network access and the right tools can monitor email and obtain login passwords.

Encryption of mail sessions

All relevant e-mail protocols have an option to encrypt the whole session. Remarkably, those options prevent a user's name and password from being sniffed, therefore they are recommended for nomadic users and whenever the internet access provider is not trusted. On sending mail, users can only control encryption at the hop from a client to its configured outgoing mail server. At any further hop, messages may be transmitted with or without encryption, depending solely on the general configuration of the transmitting server and the capabilities of the receiving one.

Encrypted mail sessions deliver messages in their original format, i.e. plain text or encrypted body, on a user's local mailbox and on the destination server's. The latter server is operated by an e-mail hosting service provider, possibly a different entity than the internet access provider currently at hand.

Encryption of the message body

There are two models for managing cryptographic keys. S/MIME employs a model based on a trusted certificate authority (CA) that signs users' public keys. OpenPGP employs a somewhat more flexible web of trust mechanism that allows users to sign one another's public keys. OpenPGP is also more flexible in the format of the messages, in that it still supports plain message encryption and signing as they used to work before MIME standardization.

In both cases, only the message body is encrypted. Headers, including originator, recipients, and subject, remain in plain text.

Standards

While popular protocols for retrieving mail include POP3 and IMAP4, sending mail is usually done using the SMTP protocol.

Another important standard supported by most e-mail clients is MIME, which is used to send binary file e-mail attachments. Attachments are files that are not part of the e-mail proper, but are sent with the e-mail.

Most e-mail clients use an X-Mailer header to identify the software used to send the message. According to RFC 2076, this is a common but non-standard header.

RFC 4409, Message Submission for Mail, details the role of the Mail submission agent.

RFC 5068, E-mail Submission Operations: Access and Accountability Requirements, provides a survey of the concepts of MTA, MSA, MDA, and MUA. It mentions that "Access Providers MUST NOT block users from accessing the external Internet using the SUBMISSION port 587" and that "MUAs SHOULD use the SUBMISSION port for message submission."

Port numbers

Email servers and client use the following TCP port numbers by default, unless configured for specialized installations:

protocol	**use**	**plain text or encrypt sessions**	**plain text sessions only**	**encrypt sessions only**
POP3	incoming mail	110		995
IMAP4	incoming mail	143		993
SMTP	outgoing mail	25		(unofficial) 465
MSA	outgoing mail	587		

Webmail

In addition to the fat client e-mail clients and small MUAs, there are also Web-based e-mail programs called webmail. Webmail has several advantages which include the ability to send and receive e-mail from anywhere using a single application: a web browser. This eliminates the need to configure an e-mail client. Significant examples of e-mail services which also provide the user a webmail interface are Hotmail, Gmail, AOL and Yahoo. The main drawbacks of webmail are that user interactions are subject to network response and that there is no offline capability (although Gmail does offer Offline Gmail through the installation of Gears). For instance, while webmail generally provides the

best experience over broadband, a fat client can provide a satisfactory experience over dialup, and messages can be searched and viewed without an internet connection. With the advent of Web 2.0 there has been a trend to do everything from within a web browser.[citation needed]

Simple Mail Transfer Protocol (**SMTP**) is an Internet standard for electronic mail (e-mail) transmission across Internet Protocol (IP) networks. SMTP was first defined in RFC 821 (STD 10), and last updated by RFC 5321 (2008) which includes the extended SMTP (ESMTP) additions, and is the protocol in widespread use today.

While electronic mail servers and other mail transfer agents use SMTP to send and receive mail messages, user-level client mail applications typically only use SMTP for sending messages to a mail server for relaying. For receiving messages, client applications usually use either the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP) to access their mail box accounts on a mail server.

The Internet Protocol Suite

Application Layer

BGP · DHCP · DNS · FTP · GTP · HTTP · IMAP · IRC · Megaco · MGCP · NNTP · NTP · POP · RIP · RPC · RTP · RTSP · SDP · SIP · **SMTP** · SNMP · SOAP · SSH · Telnet · TLS/SSL · XMPP · (more)

Transport Layer

TCP · UDP · DCCP · SCTP · RSVP · ECN · (more)

Internet Layer

IP (IPv4, IPv6) · ICMP · ICMPv6 · IGMP · IPsec · (more)

Link Layer

History

Various forms of one-to-one electronic messaging were used in the 1960s. People communicated with one another using systems developed for specific mainframe computers. As more computers were interconnected, especially in the US Government's ARPANET, standards were developed to allow users using different systems to be able to e-mail one another. SMTP grew out of these standards developed during the 1970s.

SMTP can trace its roots to two implementations described in 1971, the Mail Box Protocol, which has been disputed to actually have been implemented, but is discussed in RFC 196 and other RFCs, and the SNDMSG program, which according to RFC 2235 by Ray Tomlinson of BBN "invents" for TENEX computers the sending of mail across the ARPANET. Fewer than 50 hosts were connected to the ARPANET at this time.

Further implementations include FTP Mail and Mail Protocol, both from 1973. The work continued throughout the 1970s, until the ARPANET converted into the modern Internet around 1980. Jon Postel then proposed a Mail Transfer Protocol in 1980 that began to remove the mail's reliance on FTP. SMTP was published as RFC 821 in August 1982, also by Postel.

The SMTP standard was developed around the same time as Usenet, a one-to-many communication network with some similarities.

SMTP became widely used in the early 1980s. At the time, it was a complement to Unix to Unix Copy Program (UUCP) mail, which was better suited to handle e-mail transfers between machines that were intermittently connected. SMTP, on the other hand, works best when both the sending and receiving machines are connected to the network all the time. Both use a store and forward mechanism and are examples of push technology. Though Usenet's newsgroups are still propagated with UUCP between servers, UUCP mail has virtually disappeared along with the "bang paths" it used as message routing headers.

The article about sender rewriting contains technical background info about the early SMTP history and source routing before RFC 1123.

Sendmail was one of the first (if not the first) mail transfer agents to implement SMTP.[citation needed] Some other popular SMTP server programs include Postfix, qmail, Novell GroupWise, Exim, Novell NetMail, Microsoft Exchange Server, Sun Java System Messaging Server.

Message submission (RFC 2476) and SMTP-AUTH (RFC 2554) were introduced in 1998 and 1999, both describing new trends in e-mail delivery. Originally, SMTP servers were typically internal to an organization, receiving mail for the organization from the outside, and relaying messages from the organization to the outside. But as time went on, SMTP servers (Mail transfer agents), in practice, were expanding their roles to become message submission agents for Mail user agents, some of which were now relaying mail from the outside of an organization. (e.g. A company executive wishes to send e-mail while on a trip using the corporate SMTP server.) This issue, a consequence of the rapid expansion and popularity of the World Wide Web, meant that the SMTP protocol had to include specific rules and methods for relaying mail and authenticating users to prevent abuses such as unsolicited e-mail (spam) relaying.

As this protocol started out purely ASCII text-based, it did not deal well with binary files. Standards such as Multipurpose Internet Mail Extensions (MIME) were developed to

encode binary files for transfer through SMTP. Mail transfer agents (MTAs) developed after Sendmail also tended to be implemented 8-bit-clean, so that the alternate "just send eight" strategy could be used to transmit arbitrary data via SMTP. Non-8-bit-clean MTAs today tend to support the 8BITMIME extension, permitting binary files to be transmitted almost as easily as plain text.

Many people contributed to the core SMTP specifications, among them Jon Postel, Eric Allman, Dave Crocker, Ned Freed, Randall Gellens, John Klensin, and Keith Moore.

Protocol overview

SMTP is a relatively simple, text-based protocol, in which a mail sender communicates with a mail receiver by issuing simple command strings and supplying necessary data over a reliable ordered data stream channel, typically a Transmission Control Protocol (TCP) connection. An SMTP session consists of a series of commands, initiated by the SMTP client, and responses from the SMTP server through which the session is opened, operating parameters are exchanged, the recipients are specified, and possibly verified, and the message is transmitted, before the session is closed. The originating host is either an end-user's email client also known as mail user agent (MUA), or a relay server's mail transfer agent (MTA).

SMTP was designed as an electronic mail transport and delivery protocol, and as such it is used between SMTP systems that are operational at all times. However, it has capabilities for use as a mail submission protocol for email clients (split user-agent) that do not have the capability to operate as MTA. Such agents are also called message submission agents (MSA), sometimes also referred to as mail submission agents. They are typically end-user applications and send all messages through a smart relay server, often called the outgoing mail server, which is specified in the programs' configuration. A mail transfer agent, incorporated either in the e-mail client directly or in the relay server, typically determines the destination SMTP server by querying the Domain Name System for the mail exchanger (MX record) of each recipient's domain name. Conformant MTAs fall back to a simple address lookup (A record) of the domain name when no mail exchanger is available. In some cases an SMTP client, even a server, may also be configured to use a smart host for delivery. The SMTP client typically initiates a Transmission Control Protocol (TCP) connection to the SMTP server on the well-known port designated for SMTP, port number 25.

SMTP is a delivery protocol only. It cannot pull messages from a remote server on demand. Other protocols, such as the Post Office Protocol (POP) and the Internet Message Access Protocol (IMAP) are specifically designed for retrieving messages and managing mail boxes. However, the SMTP protocol has a feature to initiate mail queue processing on a remote server so that the requesting system may receive any messages destined for it (cf. #Remote Message Queue Starting). POP and IMAP are preferred protocols when a user's personal computer is only intermittently powered up, or Internet connectivity is only transient and hosts cannot receive message during off-line periods.

Remote Message Queue Starting

Remote Message Queue Starting is a feature of the SMTP protocol that permits a remote host to start processing of the mail queue on a server so it may receive messages destined to it by sending the TURN command. This feature however was deemed insecure and was extended in RFC 1985 with the ETRN command which operates more securely using an authentication method based on Domain Name System information.

SMTP transport example

A typical example of sending a message via SMTP to two mailboxes (alice and theboss) located in the same mail domain (example.com) is reproduced in the following session exchange.

For illustration purposes here (not part of protocol), the protocol exchanges are prefixed for the server (S:) and the client (C:).

After the message sender (SMTP client) establishes a reliable communications channel to the message receiver (SMTP server), the session is opened with a greeting by the server, usually containing its fully qualified domain name, in this case smtp.example.com. The client initiates its dialog by responding with a HELO command identifying itself in the command's parameter.

S: 220 smtp.example.com ESMTP Postfix
C: HELO relay.example.org
S: 250 Hello relay.example.org, I am glad to meet you
C: MAIL FROM:<bob@example.org>
S: 250 Ok
C: RCPT TO:<alice@example.com>
S: 250 Ok
C: RCPT TO:<theboss@example.com>
S: 250 Ok
C: DATA
S: 354 End data with <CR><LF>.<CR><LF>
C: From: "Bob Example" <bob@example.org>
C: To: Alice Example <alice@example.com>
C: Cc: theboss@example.com
C: Date: Tue, 15 Jan 2008 16:02:43 -0500
C: Subject: Test message
C:
C: Hello Alice.
C: This is a test message with 5 header fields and 4 lines in the message body.
C: Your friend,
C: Bob
C: .
S: 250 Ok: queued as 12345

C: QUIT
S: 221 Bye
{The server closes the connection}

The client notifies the receiver of the originating e-mail address of the message in a MAIL FROM command. In this example, the email message is sent to two mailboxes on the same SMTP server: one each for each recipient listed in the To and Cc header fields. The corresponding SMTP command is RCPT TO. Each successful reception and execution of a command is acknowledged by the server with a result code and response message (e.g., 250 Ok).

The transmission of the body of the mail message is initiated with a DATA command after which it is transmitted verbatim line by line and is terminated with a characteristic sequence of a new line (<CR><LF>) with just a single full stop (period) followed by another line indication (<CR><LF>).

The QUIT command ends the session.

The information that the client sends in the HELO and MAIL FROM commands are added (not seen in example code) as additional header fields to the message by the receiving server. It adds a Received and Return-Path header field, respectively.

Optional extensions

Although optional and not shown in this example, many clients ask the server for the SMTP extensions that the server supports, by using the EHLO greeting of the extended SMTP specification (RFC 1870). Clients fall back to HELO only if the server does not respond to EHLO.

Modern clients may use the ESMTP extension keyword SIZE to query the server for the maximum message size that will be accepted. Older clients and servers may try to transfer excessively-sized messages that will be rejected after consuming network resources, including connect time to network links that is paid by the minute.

Users can manually determine in advance the maximum size accepted by ESMTP servers. The client replaces the HELO command with the EHLO command.

S: 220-smtp2.example.com ESMTP Postfix
C: EHLO bob.example.org
S: 250-smtp2.example.com Hello bob.example.org [192.0.2.201]
S: 250-SIZE 14680064
S: 250-PIPELINING
S: 250 HELP

Thus smtp2.example.com declares that it will accept a fixed maximum message size no larger than 14,680,064 octets (8-bit bytes). Depending on the server's actual resource usage, it may be currently unable to accept a message this large. In the simplest case, an ESMTP server will declare a maximum SIZE with only the EHLO user interaction.

Globalization

RFC 5336 provides a UTF8STMP extension which enables SMTP support for non-ASCII email addresses, such as Pelé@live.com (simple diacritic, still mostly Latin), δοκιμή@παράδειγμα.δοκιμή, and 测试@测试.测试. Though experimental, RFC 5336 is expected to be a final standard in Fall 2009, with a Chinese version to be published in Nov 2009.

Security and spamming

One of the limitations of the original SMTP is that it has no facility for authentication of senders. Therefore the SMTP-AUTH extension was defined. However, the impracticalities of widespread SMTP-AUTH implementation and management means that E-mail spamming is not and cannot be addressed by it.

Modifying SMTP extensively, or replacing it completely, is not believed to be practical, due to the network effects of the huge installed base of SMTP. Internet Mail 2000 was one such proposal for replacement.

Spam is enabled by several factors, including vendors implementing broken MTAs (that do not adhere to standards, and therefore make it difficult for other MTAs to enforce standards), security vulnerabilities within the operating system (often exacerbated by always-on broadband connections) that allow spammers to remotely control end-user PCs and cause them to send spam, and a lack of "intelligence" in many MTAs.

There are a number of proposals for sideband protocols that will assist SMTP operation. The Anti-Spam Research Group (ASRG) of the Internet Research Task Force (IRTF) is working on a number of E-mail authentication and other proposals for providing simple source authentication that is flexible, lightweight, and scalable. Recent Internet Engineering Task Force (IETF) activities include MARID (2004) leading to two approved IETF experiments in 2005, and DomainKeys Identified Mail in 2006.

General email processing model

The overall flow for message creation, mail transport and delivery may be illustrated as follows:

> sending MUA → MSA → sending MTA → receiving MTA → MDA → receiving MUA

E-mail is submitted from an message user agent (MUA), the user's email client, to a mail server (MSA), usually using SMTP. From there, the MSA delivers the mail to an MTA, often running on the same machine. These functions may not be distinguished, or merged into one program, and a message may be directly submitted to an MTA: port 587 is used for submission to MSAs (thence to MTAs), while port 25 is used for transferring to MTAs.

The MTA looks up the destination mail exchanger records in the DNS, and relays the mail to the server on record via TCP port 25 and SMTP. Once the receiving MTA accepts the incoming message, it is delivered via a mail delivery agent (MDA) to a server which is designated for local mail delivery. The MDA either delivers the mail directly to storage, or forwards it over a network using either SMTP or the Local Mail Transfer Protocol (LMTP), a derivative of ESMTP designed for this purpose. Once delivered to the local mail server, the mail is stored for batch retrieval by authenticated mail clients (MUAs). Mail is retrieved by end-user applications, the email clients, using IMAP, a protocol that both facilitates access to mail and manages stored mail, or the Post Office Protocol (POP) which typically uses the traditional mbox mail file format. Webmail clients may use either method, but the retrieval protocol is often not a formal standard. Some local mail servers and MUAs are capable of either push or pull mail retrieval.

SMTP defines message transport, not the message content. Thus, it defines the **envelope**, such as the envelope sender, but not the headers or body of the message itself. For example, STD 10 and RFC 5321 define SMTP (the envelope), while STD 11 and RFC 5322 define the message (headers and body), formally referred to as the **Internet Message Format** (IMF).

Related Requests For Comments

- RFC 1123 – Requirements for Internet Hosts -- Application and Support (STD 3)
- RFC 1870 – SMTP Service Extension for Message Size Declaration (obsoletes: RFC 1653)
- RFC 2476 – Message Submission
- RFC 2505 – Anti-Spam Recommendations for SMTP MTAs (BCP 30)
- RFC 2920 – SMTP Service Extension for Command Pipelining (STD 60)
- RFC 3030 – SMTP Service Extensions for Transmission of Large and Binary MIME Messages
- RFC 3207 – SMTP Service Extension for Secure SMTP over Transport Layer Security (obsoletes RFC 2487)
- RFC 3461 – SMTP Service Extension for Delivery Status Notifications (obsoletes RFC 1891)
- RFC 3462 – The Multipart/Report Content Type for the Reporting of Mail System Administrative Messages (obsoletes RFC 1892)
- RFC 3463 – Enhanced Status Codes for SMTP (obsoletes RFC 1893)
- RFC 3464 – An Extensible Message Format for Delivery Status Notifications (obsoletes RFC 1894)
- RFC 3834 – Recommendations for Automatic Responses to Electronic Mail

- RFC 4409 – Message Submission for Mail (obsoletes RFC 2476)
- RFC 4952 – Overview and Framework for Internationalized E-mail
- RFC 4954 – SMTP Service Extension for Authentication (obsoletes RFC 2554)
- RFC 5068 – E-mail Submission Operations: Access and Accountability Requirements (BCP 134)
- RFC 5321 – The Simple Mail Transfer Protocol (obsoletes RFC 821 aka STD 10, RFC 974, RFC 1869, RFC 2821)
- RFC 5322 – Internet Message Format (obsoletes RFC 822 aka STD 11, and RFC 2822)
- RFC 5336 - SMTP Extension for Internationalized Email Addresses (updates RFC 2821, RFC 2822, and RFC 4952)
- RFC 5504 - Downgrading Mechanism for Email Address Internationalization

POP 3 Programs

```
import javax.mail.internet.*;
import java.util.*;
import java.io.*;

public class POP3Client {

  public static void main(String[] args) {

    Properties props = new Properties();

    String host = "utopia.poly.edu";
    String username = "eharold";
    String password = "mypassword";
    String provider = "pop3";

    try {

      // Connect to the POP3 server
      Session session = Session.getDefaultInstance(props, null);
      Store store = session.getStore(provider);
      store.connect(host, username, password);

      // Open the folder
      Folder inbox = store.getFolder("INBOX");
      if (inbox == null) {
        System.out.println("No INBOX");
```

```
        System.exit(1);
      }
      inbox.open(Folder.READ_ONLY);

      // Get the messages from the server
      Message[] messages = inbox.getMessages();
      for (int i = 0; i < messages.length; i++) {
        System.out.println("------------ Message " + (i+1)
         + " ------------");
        messages[i].writeTo(System.out);
      }

      // Close the connection
      // but don't remove the messages from the server
      inbox.close(false);
      store.close();

    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}

/**
 *  Java Network Programming, Third Edition
 *  By Elliotte Rusty Harold
 *  Third Edition October 2004
 *  ISBN: 0-596-00721-3
 */
```

Java Web Page Retrieval via a Proxy

getPage1.java retrieves the Web page http://www.cs.ait.ac.th/~ad and prints it to standard output a line at a time.

It's interesting feature is the use of the ProxyDetails class in order to work through a proxy/firewall and to deliver optional authorization details (i.e. a login and password).

If getPage1 is called in the normal way:

```
java getPage1
```

Then no proxy or authorization information is used. However, if the call is:

```
java getPage1 -proxy
```

then ProxyDetails reads in proxyInfo.txt. It should contain proxy host and port information for accessing the proxy, and an optional authorization ID (login). If the ID line is found then the class will also prompt the user for a password via a dialog box.

ProxyDetails uses Base64Converter to encode its authorization strings.

Protocol handlers

java.net.URL Architecture

It was surprising to me to realize how minimal the java.net.URL class implementation really is. A java.net.URL object instance is used to represent a URL string, where a URL string usually follows this pattern:

protocol://host:port/filepath#ref

The public accessor methods of the URL class basically reflect the private fields of a URL object, and they also provide access to the individual parts of a URL string (See Figure 1).

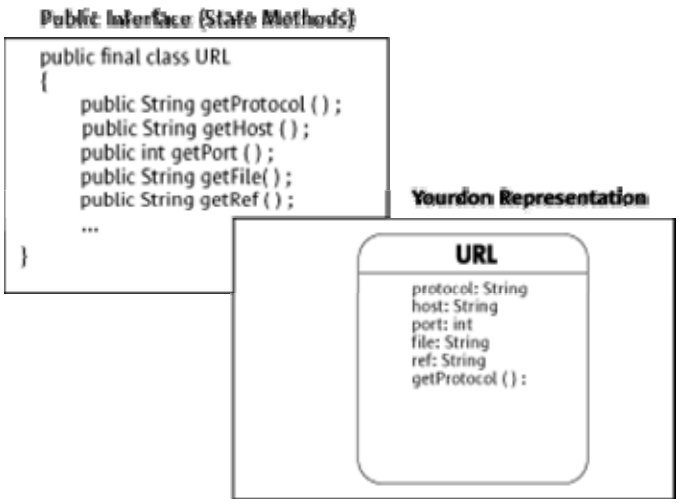


Figure 1:A URL object stores the contents of a URL string, parsed into several fields

Most of the action code in the java.net.URL class is dedicated to just the data representation in Figure 1: accessor method implementations, an equals() implementation to compare two URL objects for equivalency, a hashCode() implementation so URLs can easily be key objects in Maps, and a sameFile() method that compares two URL objects to see if everything save the "ref" field is the same.

Resolving a URL into a Resource Stream

The URL class itself doesn't actually "know "how to access the resource stream the URL represents. Instead, when a URL object is asked to resolve itself into a resource stream-by calling the URL object's openStream() method--it delegates to a protocol handler object to get the job done. The implementation of a protocol handler object translates a URL object into a client/server connection, and ultimately into a resource stream.The URL object itself doesn't perform any network requests to a server.The protocol handler does that on behalf of the URL object.

Getting a Protocol Handler Object

There is a single, stateless protocol handler object for each unique protocol string ("http ","ftp ","file ","mailto ","cvs ", and so forth.) A particular handler is delegated all responsibility for resource resolution of all URLs with the corresponding protocol string. That is, all "http:"URL objects are resolved using the HTTP protocol handler object, all "ftp:"URL objects are resolved by the FTP protocol handler and so forth.

Physically, a protocol handler is a concrete subclass of the java.net.URLStreamHandler class. The URL object stores a reference to its corresponding URLStreamHandler internally during construction. The URL object uses a cached factory object known as the URLStreamHandlerFactory to obtain the reference to the appropriate URLStreamHandler. Figure 2 illustrates how a URL object gets an initial reference to its URLStreamHandler during construction.

Note that the previous description of a URLStreamHandlerFactory as a cached factory means the URLStreamHandlerFactory creates a URLStreamHandler instance the first time it is asked for a specific protocol. Thereafter it returns a reference to the same instance whenever asked for the same protocol. The factory uses a simple mechanism for resolving protocol name into URLStreamHandler-subclass class name, which I'll describe just a little bit later in this paper.

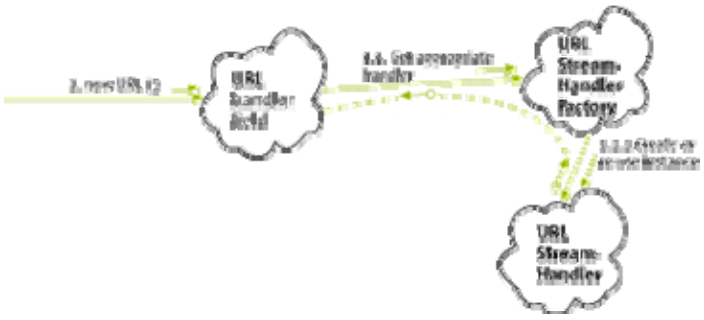


Figure 2:Sequence of interactions between java.net objects to resolve a URL into a stream

One Cause of MalformedURLExceptions

Resolving a protocol string into a protocol handler happens at construction time. This explains why you can't create a URL object with an unknown protocol--in such cases the URL constructor throws a MalformedURLException. Take a look at the following example program:

```
public class BadProtoExample
{
  public static void main(String[] args)
  {
    try {
      URL url = new URL("
            bogus://www.microsoft.com");
      System.out.println(
             "The URL: is: " + url);

     } catch (MalformedURLException mue) {
      System.err.println(mue);
    }
  }
}
```

This example program, when run on a typical Java Virtual Machine[1], produces this output on the process' standard error stream:

```
java.net.MalformedURLException:
        unknown protocol: bogus
```

The Standard URLStreamHandlers

Sun's JRE defines several standard URLStreamHandlers for popular protocols: http:, ftp:, mailto:, gopher:(?!), and even a special jar: protocol handler so you can access JAR file resources using URLs.

You can see these standard handlers, and associated implementation classes,in the JDK's RT.JAR file. Look for classes whose fully-qualified name starts with sun.net.www.protocol.For example,the class sun.net.www.protocol.http.Handler defines the HTTP protocol handler. Class sun.net.www.protocol.ftp.Handler defines the FTP protocol handler class.

These are the standard URLStreamHandler implementations. Later in this paper I'll show you how to augment the list of handlers with your own, custom URLStreamHandler class implementations, and even how to override the standard implementations.

Opening the Stream

At some point after creating a URL object, controlling code will attempt to resolve that URL into a resource stream (that is, a java.io.InputStream). The simplest controlling code just calls the URL object's openStream() method and expects to get an InputStream instance returned.

The URL object delegates all implementation of resource resolution to the URLStreamHandler and its helper classes and objects. The URLStreamHandler is stateless. An executing application may ask it to resolve several URLs into resource streams, possibly even simultaneously.

The URLStreamHandler actually creates a secondary object, known as the URLConnection, to perform an individual resource resolution. So, if a URLStreamHandler is asked to simultaneously resource 20 different URL objects into stream, all the URLStreamHandler does is create 20 different URLConnection objects, one to handle each of the 20 requests.

From an application's point of view, the URL object looks like it's doing a lot of work during the execution of this code:

```
URL url = new URL(someURLString);
InputStream resourceStream = url.openStream();
```

But in fact, the thread of execution spends very little time in actual URL class code. Instead, the URLStreamHandler and the URLConnection it creates accomplishes most of the resource resolution. The URLConnection object ultimately has the job of creating the

OutputStream object. TheURLConnection object is a task-oriented object. After creating the OutputStream object theURLConnection reaches the end of its lifecycle and can be destroyed.

Figure 3 illustrates the flow of execution for the aforementioned two lines of sample code.

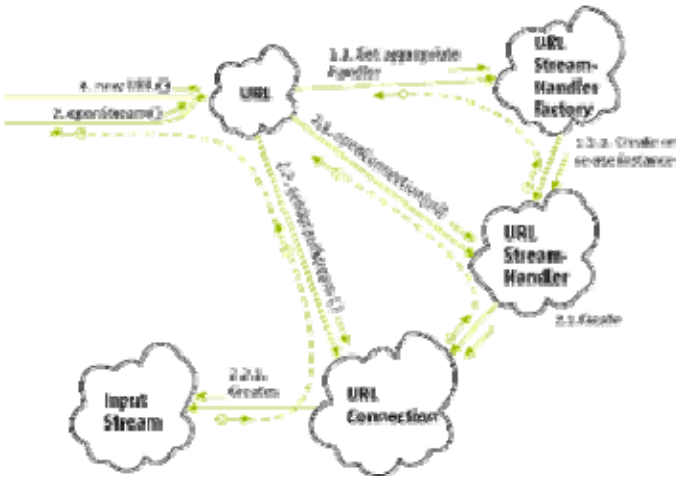


Figure 3: Flow of execution while application creates then resolves an URL into an InputStream

Figure 3 shows the URL constructor using the URLStreamHandlerFactory to obtain a URLStreamHandler. When client code later calls openStream(), the URLStreamHandler is itself used as a factory to create a URLConnection. Most of the code that establishes a connection to a server and downloads a resource stream is located in the URLConnection. The URLConnection is a single-use object whose whole life's purpose is to generate an InputStream for the client code to consume.

Class Relationships

The URL class is a state storage class. The URL object stores information about the parts of a URL string, and manages other objects in order to resolve that URL string into a resource stream when asked to do so. Figure 4 shows the static relationship between classes in the java.net.URL architecture.

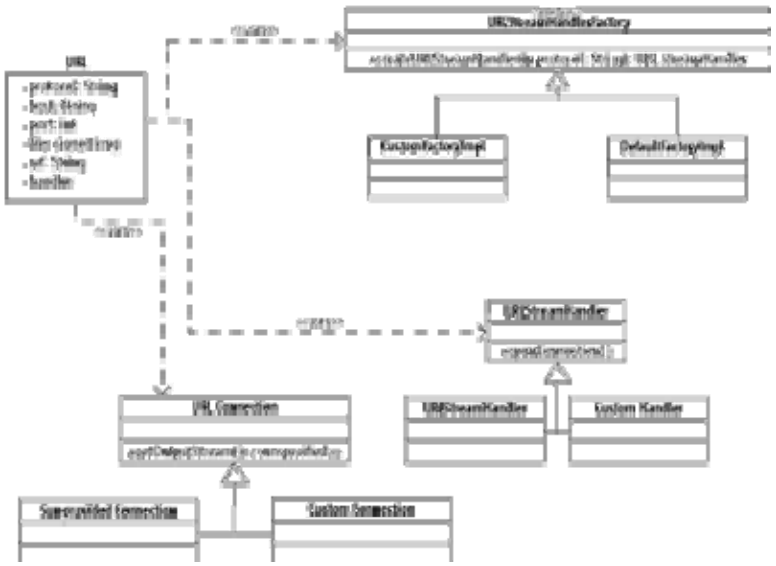


Figure 4:Static relationship between classes and implementations in UML

The URL object uses a URLStreamHandlerFactory to create a URLStreamHandler.URLStreamHandlerFactory is in fact an interface. Java provides the default implementation of the factory interface, but you can replace it with your own--in certain deployment scenarios discussed later in this paper you must provide your own factory implementation. The URL object maintains an immutable reference to its URLStreamHandler. When asked to resolve itself into a resource stream, the URL object uses its URLStreamHandler to create a URLConnection, then as part of the same code block the URL completes its use of the URLConnection. The URLConnection just represents a temporary, synchronous URLStreamHandler session. It's a place for the URLStreamHandler to place URL-object-specific state, which is necessary for stateless URLStreamHandlers.

The URLConnection is an interesting beast. Given the system as I've described it so far, the behavior of a URLConnection is only parameterizable by the URL object's state. That is, the only variables whose values differentiate the behavior of two URLConnection objects created by the same URLStreamHandler would be the field values of the URL objects (the host names, port numbers, filepaths, and ref strings stored internally by the URL objects) that create the two URLConnections. A request represented as a URL could not be augmented with any further information (such as HTTP POST data), especially because the URL class is final, so subclasses can't add state members.

I'll describe URLConnection objects in more depth later. But I'll tell you now the URL class defines a public method openConnection() (as opposed to openStream()) whose implementation returns theURLConnection object itself back to controlling code, rather than an InputStream. This allows client code to parameterize further a resource request past what the state members of the URL object can represent. For example, a URLConnection created by an http: URL generates an HTTP POST request when its public interface is used in a particular way. The contents of a single http: URL string alone can't do that.

Implementing a Custom Protocol Handler

A minimal protocol handler requires a concrete URLStreamHandler class and a concreteURLConnection class. In addition, you may need to define several helper classes that assist the URLStreamHandler and URLConnection implementations, but the bare-bones implementation requires at least those two pieces. I'm going to work through the creation of a protocol handler that accesses a local Windows registry. You can use a win32registry URL to access any Win32 registry key visible to the current user. The mapping between key names and URL strings is obvious, since the registry is just a persistent, hierarchical storage of binary, string, and integer data.

For anyone not familiar with a Windows registry, the aforementioned description pretty much states what a Windows registry is.

It is a persistent tree, basically. Nodes are named with strings. Nodes can have values, which are strings, integers or arbitrary binary data. Nodes can also have sub-nodes. The Windows registry is used to store the configuration of a host machine: operating system parameters, devices drivers and start-up driver parameters, configuration of Windows services (equivalent to Unix daemons), and the configuration of third-party software applications. It's a huge storehouse of information about the local system.

And for any reader who is outwardly hostile to anything Microsoft-related, such as a Windows registry, I can only offer that the Windows registry's hierarchical arrangement is so close to the implicitly-hierarchical arrangement of resources implied by URL trying syntax that a win32registry: protocol handler is a great example of the power of custom protocol handlers.

Normally you can only access a Windows registry in standard Java with native methods, because Microsoft only provides accessor functions in C-callable DLLs. With the win32registry: protocol handler, the protocol handler class implements all of the native code. Code that wants to access data in the registry need only reference the data with a win32registry: URL.

For example,if you wanted to know whether or not Sun's JRE was installed on the local system, you can easily find out by querying the Windows registry.JavaSoft adds registry entries storing the configuration of the JRE during JRE installation. Under the registry key, HKEY_LOCAL_MACHINE \SOFTWARE \JavaSoft \Java Runtime Environment \1.2, is all the JRE configuration parameters, such as the installation directory of the JRE (stored in the "JavaHome" subkey).

Using the win32registry: protocol handler, you can look up the value of this key with just a few lines of code:

```
URL url =
  new URL("win32-registry:///HKEY_LOCAL_MACHINE/ \
```

```
SOFTWARE/JavaSoft/Java Runtime
            Environment/1.2/JavaHome");
InputStream is = url.openStream();
DataInputStream dis = new DataInputStream(is);
String strJavaHome = dis.readUTF();
```

The code example shows that string-valued keys are encoded using UTF-encoded strings. Integer-valued keys will result in streams that include a 4-byte integer in network-byte order. Binary-valued keys will result in streams that just include the binary data of the key.[2]

Mapping URL String to Resource Addresses

As with all good software projects, the first thing you have to do is planning. One of the main services a protocol handler performs is mapping a URL string into a physical resource location. Do this by defining a convention.

The win32registry: protocol handler maps a URL string to the local registry key. The key is always on a local machine[3].

The host name of the URL string must be empty or else the protocol handler will cause a MalformedURLException to be thrown when a URL object is created. That is, all valid URLs begin with:

win32registry:///

After the third forward-slash, the win32registry: protocol handler interprets the rest of the URL string as just a registry key name (where forward-slashes are translated to the Windows-preferred blackslash). It ignores "#ref" strings.

Now that I've described how the win32registry: protocol handler will interpret and translate URL strings, I'm ready to start implementing it.

Implementing URLStreamHandler

The first thing to implement is a concrete URLStreamHandler subclass.The URLStreamHandler implementation actually only has two relatively simple tasks to perform:

1. When a new win32registry: URL is created, the URLStreamHandler must parse the URL string into the various field values host, port, file, and ref.
2. Create and return a new URLConnection instance when the URLStreamHandler's openConnection() method is called.

Parsing URL Strings

Task 1 is actually quite important. Whenever a new URL object is created, the URL constructor first obtains a reference to the appropriate URLStreamHandler (using the URLStreamHandlerFactory, as previously described), and second asks the URLStreamHandler to parse the URL string into host, port, file and ref field values. Figure 5 illustrates this two-part construction process.

Figure 5:Two-part construction of a URL object. The handler populates the URL's fields.

The URL constructor calls handler.parseURL(). Your implementation of parseURL() is supposed to parse the URL string and use the embedded values to populate the URL object's host, port, file, and ref fields. Rather than every protocol handler designer re-writing the same code to parse a URL into its constituent parts, the default implementation of parseURL() assumes the URL is in the normal protocol://host:port/file#ref format. So, if your handlers expects URLs to be of that form, you don't actually have to implement parsing code.[4] The win32registry: protocol handler expects URLs to be of the normal form, so there's no need to override the inherited parseURL() method implementation.

Creating URLConnection Instances

The second thing a concrete URLStreamHandler must do is provide a concrete implementation of the openConnection() method. This is an abstract method declared in the URLStreamHandler base class:

```
public abstract class URLStreamHandler
{
  ...
  protected abstractURLConnection
                openConnection(URL u);
  ...
}
```

The handler's openConnection() method is called directly by the URL class openStream() implementation. An openConnection() call is a request to the handler that it create an object that can resolve the URL into a resource stream. The URLConnection object is that object. The name, URLConnection, indicates that the java.net.URL architecture developers expected the only way to resolve a URL object into a stream is actually to open a connection to some server, like an HTTP or an FTP server. My win32registry: handler actually doesn't have to open a network connection to any other server. Still, I must provide aURLConnection implementation that effects the same behavior my connection implementation won't contact a different server but will instead contact the Windows operating system through a series of native calls. A deep explanation of the win32registry: URLConnection implementation follows in a moment. To complete the win32registry: handler's openConnection() implementation, all I have to do is create and return my custom URLConnection instance:

```
public class Handler
            extends java.net.URLStreamHandler
{
   protected URLConnection openConnection(URL u)
    {
      //...create and return a custom
      // URLConnection initialized
      // with a reference to the target
      // URL object...
     return new RegistryURLConnection(u);
   }
}
```

In fact, the aforementioned code is just about my entire URLStreamHandler implementation; I left most of the implementation for the RegistryURLConnection class.

Package and Class Naming Convention

One somewhat unconventional requirement of URLStreamHandler classes is that the class name and even the package name have certain restrictions. You must name the handler class Handler, as in the previous example. The package name must include the protocol name as the last dot-separated token. Any of these fully-qualified names are valid names for my win32registry: handler class:

com.develop.protocols.win32-registry.Handler
some.package.win32-registry.Handler
some.other.package.win32-registry.Handler

That is, the fully-qualified class name must end in win32registry.Handler. The reason for this unorthodox naming restriction has to do with how the URLStreamHandlerFactory maps protocol names to handler classes. Remember that when a URL is created, the URL constructor code asks the URLStreamHandlerFactory for the URLStreamHandler associated with the URL's protocol. As it turns out, the factory just uses the protocol name, combined with a well-known system property, to generate a fully-qualified class name.

The java.protocol.handler.pkgs system property is a list of package name prefixes used by the URLStreamHandlerFactory to resolve protocol names into actual handler class names. For example, imagine this property has the value com.develop.protocols. When asked to find a handler class for the win32-registry: protocol, the URLStreamHandlerFactory concatenates the system property value (com.develop.protocols) with the protocol name (win32-registry") and the expected class name "Handler". The resultant fully-qualified class name is com.develop.protocols.win32-registry.Handler, and that's the class name the factory looks for on the classpath. Figure 6 illustrates the sequence controlled by the URLStreamHandlerFactory for creating the URLStreamHandler associated with a protocol string.

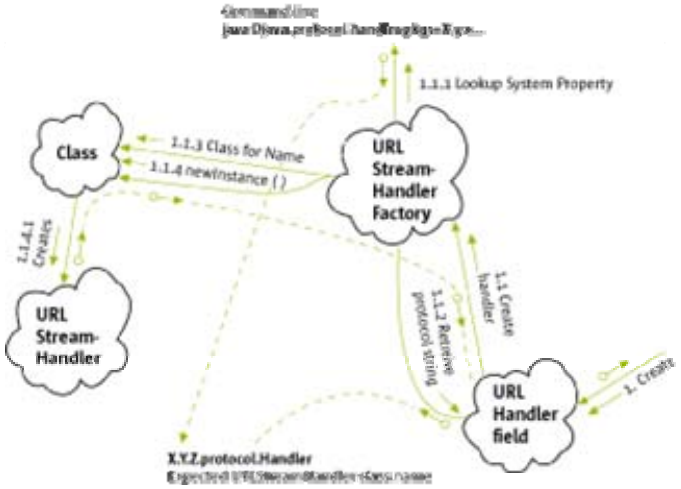


Figure 6:URLStreamHandlerFactory creates the URLStreamHandler using reflection to first load appropriate class, then create instance.

Expected URLStreamHandler class name. The java.protocol.handler.pkgs system property may contain a list of package prefixes, rather than just a single-package prefix name. The list is pipe-character ('|') separated. The URLStreamHandlerFactory evaluates each prefix in right-to-left order, stopping when it finds the first generated fully-qualified class name matching a class on the classpath.

Note also that the URLStreamHandlerFactory always appends the default protocol handler package prefix, sun.net.www.protocol, to this system property. That is, Sun's default handlers are always used if a handler can't be found in a different package first.

The win32registry: handler must be in an appropriately named package:

```
package com.develop.protocols.win32registry;
public class Handler
        extends java.net.URLStreamHandler
{
...
}
```

The VM must be started up with an appropriate value for the java.protocol.handler.pkgs system property (Type the following on one line):

```
java -Djava.protocol.handler.pkgs=
                com.develop.protocols ...
```

Implementing URLStreamHandlerFactory

An alternative to using the XYZ.protocol.Handler package and class naming convention (along with the java.protocol.handler.pkgs system property) previously described is implementing your own URLStreamHandlerFactory. The default URLStreamHandlerFactory pays attention to the java.protocol.handler.pkgs system property and applies the package and class naming conventions. Your alternative implementation of the URLStreamHandlerFactory interface could get a handler implementation from wherever it wanted.

There are a few situations where you might consider using your own factory implementation. Foremost is when you don't have access to set the java.protocol.handler.pkgs system property. This property must be set at VM start-up time, and is usually set by a command-line parameter or a start-up properties file that the VM uses. If you want to use your custom protocol as part of a component, such as a servlet, Jini service or EJBean implementation running within a container VM, then you will probably need to include a URLStreamHandlerFactory implementation as part of your component installation.

The URLStreamHandlerFactory interface is very simple:

```
public interface URLStreamHandlerFactory
{
public URLStreamHandler
     createURLStreamHandler(String protocol);
}
```

Get a URL object to use your custom factory implementation instead of the default by using an overloaded version of the URL class constructor, as in this example:

```
URL url = new URL(null,
          "cvs://server/project/folder#version",
  new MyURLStreamHandlerFactoryImpl());
```

This URL object will use MyURLStreamHandlerFactoryImpl instead of the default factory. Note, however, that Java 2 security does require a NetPermission named specifyStreamHandler granted to the calling context in order to use this constructor. That is, code without that NetPermission grant, running within a Java 2 VM with a default SecurityManager will receive an AccessControlException. The following java.policy file grant demonstrates how to assign such a permission to code signed by "MyCompany":

```
...
grant signedBy "MyCompany" {
   permission java.net.NetPermission
                  "specifyStreamHandler";
};
...
```

Implementing URLConnection

A URLConnection object manages the translation of a URL object into a resource stream. URLConnections in general can handle both interactive protocols, such as HTTP and FTP, as well as non-interactive protocols, such as "jar:" and the win32registry: protocol. That is,the URLConnection subclass used to make HTTP requests is able to handle interactive request/response dialogs with a server. The RegistryURLConnection implementation only needs to make certain local system calls to translate a URL object into a resource stream--there is no complex, multi-part request/response dialogue between the RegistryURLConnection and the native O/S.

The URLConnection class contract must be able to handle both types of resource request models. In addition, do so in a very generic fashion: the interface must be easily applied to many different types of client/server interactions. Consequently, the URLConnection class' design is somewhat abstract and rather complicated. I will explain URLConnections in general before describing my implementation of my RegistryURLConnection class in particular.

URLConnection Interface and Contract with Controlling Code

A URLConnection is a collection of request and response headers, and optionally also an OutputStream (described by request headers) and an InputStream (described by response

headers). The public API of the URLConnection class is broken down into methods that access or modify request headers, methods that access or modify response headers, and public abstract class URLConnection { //...methods to access and manipulate // request properties... public void setRequestProperty(String name, String value); public String getRequestProperty(String name); //...methods to access response properties, // known simply as "headers" in this API. public String getHeaderField(int n); public String getHeaderFieldKey(int n); public String getHeaderField(String name); //...Convenience implementations for // commonly-used header // fields. Implementations just call // getHeaderField()... public int getContentLength(); public String getContentEncoding(); public String getContentType(); ... //...get an OutputStream directly // connected to server. Only protocols // that support writing data to a server // implement this... public OutputStream getOutputStream(); //...get an InputStream directly connected // to server, contains // resource stream sent from server. Only // protocols supporting // a stream-based response from the server // implement this. public InputStream getInputStream(); ... }

All URLConnections must support the notion of request and response headers. The request headers describe the request being made, and the response headers describe the result of the request. Some protocols may also support the client sending a stream of data as part of the request. For example, an HTTP POST request includes a body that's a stream of data. A protocol that does support interaction will have an implementation of getOutputStream(). Protocols that don't support interaction (like the win32registry: protocol) will simply throw an UnknownServiceException from getOutputStream().

Similarly, some protocols support a response stream. HTTP, FTP, and my win32registry: protocol all support the concept of a response stream. In fact, any protocol that can map a URL to a response stream obviously support this concept. A counter example is the mailto: protocol. A mailto: request does not include any kind of response stream--any information you need to gather about the success of a mailto: request can be gathered from the response headers alone. Controlling code accesses the response stream of a URLConnection request using the getInputSteam() method. If the protocol doesn't support response streams, getInputStream() throws an UnknownServiceException.

When first created,the field values of the associated URL object initialize the URLConnection object. The URLConnection object has not attempted to access any resource, and hasn't opened any connection to a server. Basically, it has undergone only primary initialization. This means the field values of the associated URL object may have initialized some of the request headers.

Controlling code with a reference to the URLConnection object may call setRequestHeader() to further parameterize the request. So far in this paper, I've suggested that "controlling code" of a URLConnection instance is the openStream() implementation in the URL class. That is, "controling code" is the only block of code that can directly touch aURLConnection object. As it turns out, the URL.openConnection() method causes a new URLConnection object to be created (through the process already defined), but then simply hands the URLConnection back to the caller. This way, code

that created a URL object may have direct access to its URLConnection, and that controlling code can then manipulate the request the URLConnection is to make.The following example code shows how to perform an HTTP PORT request in Java using a http: URL's URLConnection.

```
//Application (a.k.a "controlling ")code.
URL url =new URL("http://someserver/file ");
URLConnection urlconn =url.openConnection();
//...use setRequestHeader()to modify request into
// a POST request.(The "METHOD "header is specially
// understood by the HTTP URLConnection.)...
urlconn.setRequestHeader("METHOD ","POST ");
//...Indicate the content type of the POST data...
urlconn.setContentType("
        application/x-www-form-urlencoded ");
//...equivalent to setting the
                "content-type "header...
//...Open a connection with the server...
urlconn.connect();
//...Get the output stream to send request data...
OutputStream os =urlconn.getOutputStream();
//...Write POSTed form data...
//...Get the InputStream, which completes
// the request...
InputStream response =urlconn.getInputStream();
```

Once the controlling code has initialized all necessary request headers, the controlling code calls the URLConnection's connect() method.This method has a nebulous definition in general. Basically, it means that the request headers have been fully-formed, and for the URLConnection to go ahead and make a connection to the server/service/library that is servicing the request. An HTTP URLConnection implementation would obviously make a TCP connection and send request headers in response to this method being called. The win32-registry: protocol handler goes through native methods to actually get the target registry key value in response to this method. Note that getOutputStream() and getInputStream() methods both call connect() initially if it hasn't been called yet.

When connect()returns, the response headers should be initialized to values indicating the result of the request.

Controlling code expecting a response stream would call getInputStream() at this point. The contents of the stream and the response header values indicate the result of the URL request. Again, the URLConnection base class has a rather generic definition, meant to be a broad umbrella under which the concepts of parameterized requests and responses in general can fit. When implementing a protocol handler you must understand how your

protocol fits into this generic model so that you can implement a meaningful URLConnection class. I recommend you read the JavaDocs for the java.net.URLConnection class. There are several convenience and hook methods in the API I left out or glossed over in order to discuss the essential points.

The RegistryURLConnection Implementation

After that lengthy description of the URLConnection class, I think you'll be surprised to see how simple the RegistryURLConnection implementation looks. After all, my win32registry: protocol doesn't support an interactive OutputStream, and it can ignore any request header settings made by controlling code.That is, because all information needed to find a resource is located in the URL object's fields, there's no need for my implementation to pay attention to further parameterization by controlling code. The setRequestHeader () implementation is empty:

```
package com.develop.protocols;
public class RegistryURLConnection
                extends java.net.URLConnection
{
  ...
  public void setRequestHeader(String name,
                          String value) { }
  ...
}
```

Meaning of RegistryURL Connection's Response Headers

My implementation does have to do a little cognitive mapping to squeeze a registry value response into the URLConnection's response headers and OutputStream. First of all, RegistryURLConnection.connect() sets a single response header called "Exists" to indicate whether or not the target registry key exists. The valid values will be "True "or "False."

Next, if a registry key does exist,that doesn't mean it has a value. For example, the \HKEY_LOCAL_MACHINE \SOFT- WARE \JavaSoft \Java Runtime Environment \1.2 key doesn't have a value, it's just a parent node to sub-nodes that actually do have values. A request for this node would result in a value of "True "for the "Exists "response header, but a value of "application/x-null "for the "Content-type "response header.

The other valid "Content-type "header field value," application/x-java-DataStream,"is used if the key does exist and it does have a value associated with it. In this case, another

header field, "DataType "indicates the registry key's value. This response header will have one of three values: UTF, int, or binary.

In the case of binary, the Content-length header field indicates the length of the response stream. The implementation of all the getXYZ()methods to access response headers forward the call to the getHeaderField() method. The getHeaderField() implementation just accesses values in a Map object. This is a minimal implementation of response headers that supports actually having response headers (as opposed to a null implementation, used by protocol handlers that don't use response headers at all).

Creating Content and Protocol Handlers in Java, Part 2

Part 1 of this series showed you how to work with an existing content handler and how to create a new one. It also discussed how to define a new MIME type for a new kind of file (the .xy file) and how to transfer it from a simple HTTP server to a Java client. In principle, the Internet browser (Internet Explorer 6.0) and the Java client that you have developed for the new content handler use the same idea—that is, both of them use the HTTP specification protocol for connecting and sending a request to the HTTP server HTTPServer.java (the source code for this minimal HTTP server can be viewed in Part 1). When they get the sever response, they process it using their appropriate content handlers.

The question is: how does the Java client (or the browser) know that it will "talk" with an HTTP server and how does it know what to "say"? The answer is that the Java client has no idea what kind of server it will "talk" to. The Java client just parses the provided URL and extracts the protocol name (http, ftp, etc.). After that, it associates the protocol name with the corresponding protocol implementation. These tasks are accomplished by the protocol handler.

In this article, you will learn how to work with an existing Java protocol handler and how to create a new one. Of course, when you're writing your own protocol handler, it will not be recognized by Internet servers, or any other kind of servers, because they can't possibly recognize it. For testing purposes, these examples use the HTTPServer.java you developed in Part 1. This server will recognize the protocol handler defined in Part 2 and the content handler defined in Part 1.

Working with an Existing Protocol Handler

Whenever you "ask" for an Internet resource, the browser must complete three essential tasks.

1. The first task is to parse the specified URL (conforming to URL specifications) and extract the protocol name.
2. The second task is based on the protocol name—the browser is choosing "the right words" for connecting and communicating with the requested server. The

final task is to put the received resource into a readable form. These first two tasks are accomplished by the protocol handler.

3. The final task is accomplished by the content handler. The browser has a protocol handler for every kind of protocol, so the HTTP protocol handler know how to "talk" with an HTTP server, a FTP protocol handler know how to "talk" with an FTP server, etc.

In Java, URL objects use protocol handlers to open connections to specified servers. By default, Java supports a set of protocol handlers that have been developed for the most common protocols, but Java also offers you support for any new protocol handlers you should create. Such support can be found in the java.net.URLStreamHandler and java.net.URLConnection classes.

The default protocol handlers' class names, along with their associated package names, reflect the protocol names themselves. For example, the http protocol's proper protocol handler is in the sun.net.www.protocol.http package and its name is Handler.class. The sun.net.www.protocol path is fixed and the rest depends on the protocol name. The table below contains the correspondence between some protocol names and their protocol handlers (J2SE 1.5):

Protocol name	path	Class
doc	sun.net.www.protocol.doc	Handler.class
file	sun.net.www.protocol.file	Handler.class
ftp	sun.net.www.protocol.ftp	Handler.class
gopher	sun.net.www.protocol.gopher	Handler.class
http	sun.net.www.protocol.http	Handler.class
jar	sun.net.www.protocol.jar	Handler.class
mailto	sun.net.www.protocol.mailto	Handler.class
netdoc	sun.net.www.protocol.netdoc	Handler.class
systemresource	sun.net.www.protocol.systemresorce	Handler.class
verbatim	sun.net.www.protocol.verbatim	Handler.class
Xy (your protocol handler)	**sun.net.www.protocol.xy**	**Handler.class**

Java applet

A **Java applet** is an applet delivered to the users in the form of Java bytecode. Java applets can run in a Web browser using a Java Virtual Machine (JVM), or in Sun's AppletViewer, a stand-alone tool for testing applets. Java applets were introduced in the first version of the Java language in 1995. Java applets are usually written in the Java programming language but they can also be written in other languages that compile to Java bytecode such as Jython.

Applets are used to provide interactive features to web applications that cannot be provided by HTML. Since Java's bytecode is platform independent, Java applets can be executed by browsers for many platforms, including Windows, Unix, Mac OS and Linux.

There are open source tools like applet2app which can be used to convert an applet to a stand alone Java application/windows executable/linux executable. This has the advantage of running a Java applet in offline mode without the need for internet browser software.

Many influential Java developers, blogs and magazines are recommending that the Java Web Start technology be used in place of Applets .

A Java Servlet is sometimes informally compared to be "like" a server-side applet, but it is different in its language, functions, and in each of the characteristics described here about applets.

Technical information

Java applets are executed in a sandbox by most web browsers, preventing them from accessing local data. The code of the applet is downloaded from a web server and the browser either embeds the applet into a web page or opens a new window showing the applet's user interface. The applet can be displayed on the web page by making use of the deprecated applet HTML element , or the recommended object element . This specifies the applet's source and the applet's location statistics.

A Java applet extends the class java.applet.Applet, or in the case of a Swing applet, javax.swing.JApplet. The class must override methods from the applet class to set up a user interface inside itself (Applet is a descendant of Panel which is a descendant of Container).

Advantages

A Java applet can have any or all of the following advantages:

- it is simple to make it work on Linux, Windows and Mac OS i.e. to make it cross platform
- the same applet can work on "all" installed versions of Java at the same time, rather than just the latest plug-in version only. However, if an applet requires a later version of the JRE the client will be forced to wait during the large download.
- it is supported by most web browsers
- it will cache in most web browsers, so will be quick to load when returning to a web page but may get stuck in the cache and have issues when new versions come out.
- it can have full access to the machine it is running on if the user agrees
- it can improve with use: after a first applet is run, the JVM is already running and starts quickly, benefitting regular users of Java but the JVM will need to restart each time the browser starts fresh.
- it can run at a speed that is comparable to (but generally slower than) other compiled languages such as C++, but many times faster than JavaScript

- it can move the work from the server to the client, making a web solution more scalable with the number of users/clients
- developers can develop and debug an applet direct simply by creating a main routine (either in the applet's class or in a separate class) and call init() and start() on the applet, thus allowing for development in their favorite J2SE development environment. All one has to do after that is re-test the applet in the appletviewer program or a web browser to ensure it conforms to security restrictions.

Disadvantages

A Java applet may have any of the following disadvantages:

- It requires the Java plug-in, which isn't available by default on all web browsers.
- Prior to version 6u12, Sun did not provide a 64-bit version of its Java plug-in, forcing users to use the 32-bit plugin with a 32-bit browser.[1]
- It cannot start until the Java Virtual Machine is running, and this may have significant startup time the first time it is used.
- If untrusted, it has severely limited access to the user's system - in particular having no direct access to the client's disk or clipboard (although some would argue that this is a security benefit instead of a disadvantage, as ad-hoc unrestricted access to a client's disk would be incredibly dangerous).
- Some organizations only allow software installed by the administrators. As a result, many users cannot view applets by default.
- Applets may require a specific JRE.[2]

Compatibility issues

Sun has made a considerable effort to ensure compatibility is maintained between Java versions as they evolve. For example, Microsoft's Internet Explorer, the most popular web browser since the late 1990s[citation needed], used to ship with the Microsoft Java Virtual Machine as the default. The MSJVM had some extra non-Java features added which, if used, would prevent MSJVM applets from running on Sun's Java (but not the other way round).[citation needed] Sun sued for breach of trademark, as the point of Java was that there should be no proprietary extensions and that code should work everywhere.[citation needed] Development of MSJVM was frozen by a legal settlement, leaving many users with an extremely outdated Java virtual machine. Later, in October 2001, MS stopped including Java with Windows, and for some years it has been left to the computer manufacturers to ship Java independently of the OS.[citation needed]

Some browsers (notably Netscape) do not do a good job of handling height=100% on applets which makes it difficult to make an applet fill most of the browser window (JavaScript can, with difficulty, be used for this). Having the applet create its own main window is not a good solution either, as this leaves the browser window as a largely useless extra window and leads to a large chance of the applet being terminated unintentionally by the user closing the parent browser window.[citation needed]

Image handling

```
import java.io.*;
import java.net.URL;
import java.util.*;
import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;
class FileAndTextTransferHandler extends TransferHandler {
private DataFlavor fileFlavor, stringFlavor;
private ThumbnailImagesPanelNew tip;
private ImagesNamesPanel inp;
private JPanel source;
private boolean shouldRemove;
protected String newline = "\n";
int x=30,y=20,previousSize=0;
java.io.FileFilter fileFilter=new FileListFilter();
/** Creates a new instance of FileAndTextTransferHandler */
public FileAndTextTransferHandler()
{
fileFlavor = DataFlavor.javaFileListFlavor;
stringFlavor = DataFlavor.stringFlavor;
x=30;
y=30;
}
/**
* Methode Name : setDropTraget()
* Parameter : Object
* Return Type : void
* Purpose : This method sets the drop trget
*/
public void setDropTraget(Object component)
{
if(component instanceof ThumbnailImagesPanelNew)
{
System.out.println("Thumbnails");
tip=(ThumbnailImagesPanelNew)component;
source=tip.getPanel();
```

```
}
else
{
System.out.println("Names List");
inp=(ImagesNamesPanel)component;
source=(JPanel)inp;
}
}
public boolean importData(JComponent c, Transferable t)
{
if (!canImport(c, t.getTransferDataFlavors()))
{
return false;
}
try
{
if (hasFileFlavor(t.getTransferDataFlavors()))
{
String str = null;
java.util.List files =(java.util.List)t.getTransferData(fileFlavor);
//Collections.sort(files, String.CASE_INSENSITIVE_ORDER);
tip.storeLocalImages(files);
java.util.List fileList= getAllFiles(files);
tip.storeLocalImages(fileList);
tip.displayThumbnails();
inp.addLocalImgPath(files);
System.out.println("<<<<<< Inside Method importData( >>>>>>>>>>>>>>");
return true;
}
}
catch(Exception ex)
{
ex.printStackTrace();
}
return false;
}
public int getSourceActions(JComponent c)
{
return COPY_OR_MOVE;
}
public boolean canImport(JComponent c, DataFlavor[] flavors)
{
if (hasFileFlavor(flavors))
{
return true;
}
```

```
return false;
}
private boolean hasFileFlavor(DataFlavor[] flavors)
{
for (int i = 0; i < flavors.length; i++)
\{
if (fileFlavor.equals(flavors[i]))
{
return true;
}
}
return false;
}
private boolean hasStringFlavor(DataFlavor[] flavors)
{
for (int i = 0; i < flavors.length; i++)
{
if (stringFlavor.equals(flavors[i]))
{
return true;
}
}
return false;
}

public java.util.List getAllFiles(java.util.List fileList)
{
Iterator it=fileList.iterator();
java.util.ArrayList allDirFiles=null;
java.util.List filesList=new ArrayList();
while(it.hasNext())
{
File f=(File)it.next();
System.out.println("File Name==="+f.getName());
if(f.isDirectory())
{
filesList.add(f);// Directory Added
allDirFiles=ListDirectory.getAllFilesList(f);
}
else
{
filesList.add(f);
}
if(allDirFiles!=null)
{
Iterator it2=allDirFiles.iterator();
```

```
while(it2.hasNext())
{
filesList.add((File)it2.next());
}
it2=null;
}
ListDirectory.clearList();
}

return filesList;
}


public File[] directoryContent(File Dir)
{
File []files=Dir.listFiles(fileFilter);
for(int i=0;i<files.length;i++)
{
if(files[i].isDirectory())
{
directoryContent(files[i]);
}
}
return files;
}



}

class ListDirectory
{
static int indentLevel = -1;
static ArrayList allFilesList=new ArrayList();
static void listPath(File path)
{
File files[]; // list of files in a directory
indentLevel++; // going down...
// Create list of files in this dir.
files = path.listFiles(new FileListFilter());
// Sort with help of Collections API.
Arrays.sort(files);
for (int i=0, n=files.length; i < n; i++)
{
for (int indent=0; indent < indentLevel; indent++)
{
```

```
//System.out.print(" ");
}
if(!files[i].isDirectory())
{
////System.out.println(files[i].getPath());
allFilesList.add(files[i]);
}
if (files[i].isDirectory())
{
// Recursively descend dir tree.
listPath(files[i]);
}
}
indentLevel--; // And going up
}
public static ArrayList getAllFilesList(File path)
{
listPath(path);
return allFilesList;
}
public static void clearList()
{
allFilesList.clear();:eek:

}
}
```

RMI

The **Java Remote Method Invocation** API, or **Java RMI**, a Java application programming interface, performs the object-oriented equivalent of remote procedure calls.

Two common implementations of the API exist:

1. The original implementation depends on Java Virtual Machine (JVM) class representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as Java Remote Method Protocol (JRMP).
2. In order to support code running in a non-JVM context, a CORBA version was later developed.

Usage of the term **RMI** may denote solely the programming interface or may signify both the API and JRMP, whereas the term RMI-IIOP (read: RMI over IIOP) denotes the RMI interface delegating most of the functionality to the supporting CORBA implementation.

The programmers of the original RMI API generalized the code somewhat to support different implementations, such as an HTTP transport. Additionally, the ability to pass arguments "by value" was added to CORBA in order to support the RMI interface. Still, the RMI-IIOP and JRMP implementations do not have fully identical interfaces.

RMI functionality comes in the package java.rmi, while most of Sun's implementation is located in the sun.rmi package. Note that with Java versions before Java 5.0 developers had to compile RMI stubs in a separate compilation step using **rmic**. Version 5.0 of Java and beyond no longer require this step.

Summary::

The Java language provides a powerful addition to the tools that programmers have at their disposal. Java makes programming easier because it is object-oriented and has automatic garbage collection. In addition, because compiled Java code is architecture-neutral, Java applications are ideal for a diverse environment like the Internet.

For more information, visit "*http://java.sun.com/*".

Key words::

- Java
- RMI
- UUCP,UDP,TCP/IP
- Servlet,JSP
- POP3 ,SMTP,FTP,Socket

Key term quiz:

1. OOPS is defined as----------.

2. Data encapsulation----------.

3. Inheritance means----------.

4. In java ---------- is the process of linking a method which is in other language at run time?

5. Casting does ----------.

6. ----------is a collection of classes and interfaces that provides a high-level layer of access protection and name space management.

7. MARKER interface.means ----------.

8. ----------is a growable array of objects and dynamic.

9. JDBC is a set ----------.

10. SSI is expanded as---------

Review Questions::

1. All remote objects must extend ----------

a UnicastRemoteObject

b. UnicastObject

c. MulticastRemoteObject

d.All of the above

2. TCP/IP is ---------- between the client and the server and it is a reliable and there is a confirmation regarding reaching the message to the destination

a a two-way communication

b. one way communication

c. secured communication

d. none of the above

3. ----------. reflect package has the ability to analyze itself in runtime.

a java. lang

b. java.io

c. java.awt

d. none of the above

4. Super () can be used to invoke a ----------constructor.

a super class

b. both super and sub classes

c. sub class

d. public member of a class

5. ----------class cannot be subclassed and it prevents other programmers from subclassing a secure class to invoke insecure methods

a Super

b. Abstract

c. Sub

d. A final

Review Questions::

In your own words briefly answer the following

1**.** What is the difference between procedural and object-oriented programs?

2. What are Encapsulation, Inheritance and Polymorphism?-

3. What is the difference between constructor and method?

4. What is the use of bin and lib in JDK?

5. What is final, finalize () and finally**?**

6. What is method overloading and method overriding?

7. What is interface and its use?

8. What is an abstract class?

9. What is the difference between String and String Buffer?

10**.** What is daemon thread and which method is used to create the daemon thread?

Lab Exercise::

Complete the following exercise

1. Create a java program to get date from the server using TCP
2. Create a java program to get date from the server using UDP
3. Develop an application for FTP Server through which we can download and Upload files.