Java sockets 101

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

Sockets, which provide a mechanism for communication between two computers, have been around since long before the Java language was a glimmer in James Gosling's eye. The language simply lets you use sockets effectively without having to know the details of the underlying operating system. Most books that focus on Java coding either fail to cover the topic, or leave a lot to the imagination. This tutorial will tell you what you really need to know to start using sockets effectively in your Java code. Specifically, we'll cover:

* What sockets are
* Where they fit into the structure of programs you're likely to write
* The simplest sockets implementation that could possibly work -- to help you understand the basics
* A detailed walkthrough of two additional examples that explore sockets in multithreaded and pooled environments
* A brief discussion of an application for sockets in the real world

If you can describe how to use the classes in the `java.net` package, this tutorial is probably a little basic for you, although it might be a good refresher. If you have been working with sockets on PCs and other platforms for years, the initial sections might bore you. But if you are new to sockets, and simply want to know what they are and how to use them effectively in your Java code, this tutorial is a great place to start.

Getting help

For questions about the content of this tutorial, contact the authors, Roy Miller (at *rmiller@rolemodelsoft.com*) or Adam Williams (at *awilliams@rolemodelsoft.com*).

Roy Miller and Adam Williams are Software Developers at RoleModel Software, Inc. They have worked jointly to prototype a socket-based application for the TINI Java platform from Dallas Semiconductor. Roy and Adam are currently working on porting a COBOL financial transaction system to the Java platform, using sockets.

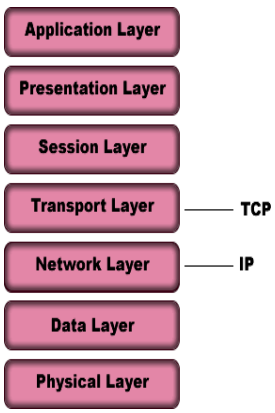
Prior to joining RoleModel, Roy spent six years with Andersen Consulting (now Accenture) developing software and managing projects. He co-authored *Extreme Programming Applied: Playing to Win* (Addison-Wesley XP Series) scheduled for publication in October 2001.

Section 2. Socket basics

Introduction

Most programmers, whether they're coding in the Java language or not, don't want to know much about low-level details of how applications on different computers communicate with each other. Programmers want to deal with higher-level abstractions that are easier to understand. Java programmers want objects that they can interact with via an intuitive interface, using the Java constructs with which they are familiar.

Sockets live in both worlds -- the low-level details that we'd rather avoid and the abstract layer we'd rather deal with. This section will explore just enough of the low-level details to make the abstract application understandable.



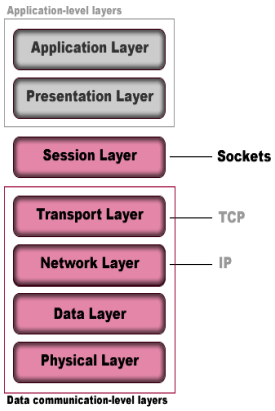
Computer networking 101

Computers operate and communicate with one another in a very simple way. Computer chips are a collection of on-off switches that store and transmit data in the form of 1s and 0s. When computers want to share data, all they need to do is stream a few million of these bits and bytes back and forth, while agreeing on speed, sequence, timing, and such. How would you like to worry about those details every time you wanted to communicate information between two applications?

To avoid that, we need a set of packaged protocols that can do the job the same way every time. That would allow us to handle our application-level work without having to worry about the low-level networking details. These sets of packaged protocols are called *stacks*. The most common stack these days is TCP/IP. Most stacks (including TCP/IP) adhere roughly to the International Standards Organization (ISO) Open Systems Interconnect Reference Model (OSIRM). The OSIRM says that there are seven logical layers in

a reliable framework for computer networking (see the diagram). Companies all over have contributed something that implements some of the layers in this model, from generating the electrical signals (pulses of light, radio frequency, and so on) to presenting the data to applications. TCP/IP maps to two layers in the OSI model, as shown in the diagram.

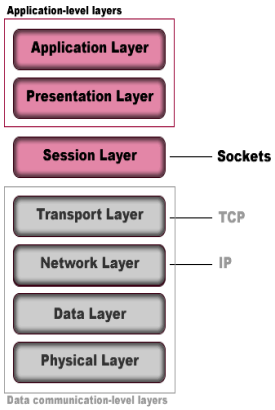
We won't go into the details of the layers too much, but we want you to be aware of where sockets fit.



Where sockets fit

Sockets reside roughly at the *Session Layer* of the OSI model (see the diagram). The Session Layer is sandwiched between the application-oriented upper layers and the real-time data communication lower layers. The Session Layer provides services for managing and controlling data flow between two computers. As part of this layer, sockets provide an abstraction that hides the complexities of getting the bits and bytes on the wire for transmission. In other words, sockets allow us to transmit data by having our application indicate that it wants to send some bytes. Sockets mask the nuts and bolts of getting the job done.

When you pick up your telephone, you provide sound waves to a sensor that converts your voice into electrically transmittable data. The phone is a human's interface to the telecommunications network. You aren't required to know the details of how your voice is transported, only the party to whom you would like to connect. In the same sense, a socket acts as a high-level interface that hides the complexities of transmitting 1s and 0s across unknown channels.



Exposing sockets to an application

When you write code that uses sockets, that code does work at the *Presentation Layer*. The Presentation Layer provides a common representation of information that the *Application Layer* can use. Say you are planning to connect your application to a legacy banking system that understands only EBCDIC. Your application domain objects store information in ASCII format. In this case, you are responsible for writing code at the Presentation Layer to convert data from EBCDIC to ASCII, and then (for example) to provide a domain object to your Application Layer. Your Application Layer can then do whatever it wants with the domain object.

The socket-handling code you write lives only at the Presentation Layer. Your Application Layer doesn't have to know anything about how sockets work.

What are sockets?

Now that we know the role sockets play, the question remains: What is a socket? Bruce Eckel describes a socket this way in his book *Thinking in Java*:

The socket is the software abstraction used to represent the "terminals" of a connection between two machines. For a given connection, there's a socket on each machine, and you can imagine a hypothetical "cable" running between the two machines with each end of the "cable" plugged into a socket. Of course, the physical hardware and cabling between machines is completely unknown. The whole point of the abstraction is that we don't have to know more than is necessary.

In a nutshell, a socket on one computer that talks to a socket on another computer creates a communication channel. A programmer can use that channel to send data between the two machines. When you send data, each layer of the TCP/IP stack adds appropriate header information to wrap your data. These headers help the stack get your data to its destination. The good news is that the Java language hides all of this from you by providing the data to your code on streams, which is why they are sometimes called *streaming sockets*.

Think of sockets as handsets on either side of a telephone call -- you and I talk and listen on our handsets on a dedicated channel. The conversation doesn't end until we decide to hang up (unless we're using cell phones). And until we hang up, our respective phone lines are busy.

If you need to communicate between two computers without the overhead of higher-level mechanisms like ORBs (and CORBA, RMI, IIOP, and so on), sockets are for you. The low-level details of sockets get rather involved. Fortunately, the Java platform gives you

some simple yet powerful higher-level abstractions that make creating and using sockets easy.

Types of sockets

Generally speaking, sockets come in two flavors in the Java language:

* TCP sockets (implemented by the `Socket` class, which we'll discuss later)
* UDP sockets (implemented by the `DatagramSocket` class)

TCP and UDP play the same role, but they do it differently. Both receive transport protocol packets and pass along their contents to the Presentation Layer. TCP divides messages into packets (*datagrams*) and reassembles them in the correct sequence at the receiving end. It also handles requesting retransmission of missing packets. With TCP, the upper-level layers have much less to worry about. UDP doesn't provide these assembly and retransmission requesting features. It simply passes packets along. The upper layers have to make sure that the message is complete and assembled in correct sequence.

In general, UDP imposes lower performance overhead on your application, but only if your application doesn't exchange lots of data all at once and doesn't have to reassemble lots of datagrams to complete a message. Otherwise, TCP is the simplest and probably most efficient choice.

Because most readers are more likely to use TCP than UDP, we'll limit our discussion to the TCP-oriented classes in the Java language.

Section 3. An undercover socket

Introduction

The Java platform provides implementations of sockets in the `java.net` package. In this tutorial, we'll be working with the following three classes in `java.net`:

* `URLConnection`
* `Socket`
* `ServerSocket`

There are more classes in `java.net`, but these are the ones you'll run across the most often. Let's begin with `URLConnection`. This class provides a way to use sockets in your Java code without having to know *any* of the underlying socket details.

Using sockets without even trying

The `URLConnection` class is the abstract superclass of all classes that create a communications link between an application and a URL. `URLConnections` are most useful for getting documents on Web servers, but can be used to connect to any resource identified by a URL. Instances of this class can be used both to read from and to write to the resource. For example, you could connect to a servlet and send a well-formed XML `String` to the server for processing. Concrete subclasses of `URLConnection` (such as `HttpURLConnection`) provide extra features specific to their implementation. For our example, we're not doing anything special, so we'll make use of the default behaviors provided by `URLConnection` itself.

Connecting to a URL involves several steps:

* Create the `URLConnection`
* Configure it using various setter methods
* Connect to the URL
* Interact with it using various getter methods

Next, we'll look at some sample code that demonstrates how to use a `URLConnection` to request a document from a server.

The URLClient class

We'll begin with the structure for the `URLClient` class.

```
import java.io.*;
import java.net.*;
public class URLClient {
    protected URLConnection connection;
    public static void main(String[] args) {
    }
    public String getDocumentAt(String urlString) {
    }
}
```

The first order of business is to import `java.net` and `java.io`.

We give our class one instance variable to hold a `URLConnection`.

Our class has a `main()` method that handles the logic flow of surfing for a document. Our class also has a `getDocumentAt()` method that connects to the server and asks it for the given document. We will go into the details of each of these methods next.

Surfing for a document

The `main()` method handles the logic flow of surfing for a document:

```
public static void main(String[] args) {
    URLClient client = new URLClient();
    String yahoo = client.getDocumentAt("http://www.yahoo.com");
    System.out.println(yahoo);
}
```

Our `main()` method simply creates a new `URLClient` and calls `getDocumentAt()` with a valid URL `String`. When that call returns the document, we store it in a `String` and then print it out to the console. The real work, though, gets done in the `getDocumentAt()` method.

Requesting a document from a server

The `getDocumentAt()` method handles the real work of getting a document over the Web:

```
public String getDocumentAt(String urlString) {
    StringBuffer document = new StringBuffer();
    try {
        URL url = new URL(urlString);
        URLConnection conn = url.openConnection();
        BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputS
        String line = null;
        while ((line = reader.readLine()) != null)
              document.append(line + "\n");
        reader.close();
    } catch (MalformedURLException e) {
        System.out.println("Unable to connect to URL: " + urlString);
    } catch (IOException e) {
        System.out.println("IOException when connecting to URL: " + urlString);
    }
    return document.toString();
}
```

The `getDocumentAt()` method takes a `String` containing the URL of the document we want to get. We start by creating a `StringBuffer` to hold the lines of the document. Next, we create a new `URL` with the `urlString` we passed in. Then we create a `URLConnection` and open it:

```
URLConnection conn = url.openConnection();
```

Once we have a `URLConnection`, we get its `InputStream` and wrap it in an `InputStreamReader`, which we then wrap in a `BufferedReader` so that we can read lines of the document we're getting from the server. We'll use this wrapping technique often when dealing with sockets in Java code, but we won't always discuss it in detail. You should be familiar with it before we move on:

```
BufferedReader reader =
     new BufferedReader(new InputStreamReader(conn.getInputStream()));
```

Having our `BufferedReader` makes reading the contents of our document easy. We call `readLine()` on `reader` in a `while` loop:

```
String line = null;
while ((line = reader.readLine()) != null)
    document.append(line + "\n");
```

The call to `readLine()` is going to *block* until in reaches a line termination character (for example, a newline character) in the incoming bytes on the `InputStream`. If it doesn't get one, it will keep waiting. It will return `null` only when the connection is closed. In this case, once we get a line, we append it to the `StringBuffer` called `document`, along with a newline character. This preserves the format of the document that was read on the server side.

When we're done reading lines, we close the `BufferedReader`:

```
reader.close();
```

If the `urlString` supplied to a `URL` constructor is invalid, a `MalformedURLException` is thrown. If something else goes wrong, such as when getting the `InputStream` on the connection, an `IOException` is thrown.

Wrapping up

Beneath the covers, `URLConnection` uses a socket to read from the URL we specified (which just resolves to an IP address), but we don't have to know about it and we don't care. But there's more to the story; we'll get to that shortly.

Before we move on, let's review the steps to create and use a `URLConnection`:

1. Instantiate a `URL` with a valid URL `String` of the resource you're connecting to (throws a `MalformedURLException` if there's a problem).

2. Open a connection on that `URL`.

3. Wrap the `InputStream` for that connection in a `BufferedReader` so you can read lines.

4. Read the document using your `BufferedReader`.

5. Close your `BufferedReader`.

You can find the complete code listing for `URLClient` at Code listing for URLClient on page 33.

Section 4. A simple example

Background

The example we'll cover in this section illustrates how you can use `Socket` and `ServerSocket` in your Java code. The client uses a `Socket` to connect to a server. The server listens on port 3000 with a `ServerSocket`. The client requests the contents of a file on the server's C: drive.

For the sake of clarity, we split the example into the client side and the server side. At the end, we'll put it all together so you can see the entire picture.

We developed this code in IBM VisualAge for Java 3.5, which uses JDK 1.2. To create this example for yourself, JDK 1.1.7 or greater should be fine. The client and the server will run on a single machine, so don't worry about having a network available.

Creating the RemoteFileClient class

Here is the structure for the `RemoteFileClient` class:

```
import java.io.*;
import java.net.*;
public class RemoteFileClient {
    protected String hostIp;
    protected int hostPort;
    protected BufferedReader socketReader;
    protected PrintWriter socketWriter;
    public RemoteFileClient(String aHostIp, int aHostPort) {
        hostIp = aHostIp;
        hostPort = aHostPort;
    }
    public static void main(String[] args) {
    }
    public void setUpConnection() {
    }
    public String getFile(String fileNameToGet) {
    }
    public void tearDownConnection() {
    }
}
```

First we import `java.net` and `java.io`. The `java.net` package gives you the socket tools you need. The `java.io` package gives you tools to read and write streams, which is the only way you can communicate with TCP sockets.

We give our class instance variables to support reading from and writing to socket streams, and to store details of the remote host to which we will connect.

The constructor for our class takes an IP address and a port number for a remote host and assigns them to instance variables.

Our class has a `main()` method and three other methods. We'll go into the details of these methods later. For now, just know that `setUpConnection()` will connect to the remote

server, `getFile()` will ask the remote server for the contents of `fileNameToGet`, and `tearDownConnection()` will disconnect from the remote server.

Implementing main()

Here we implement the `main()` method, which will create the `RemoteFileClient`, use it to get the contents of a remote file, and then print the result:

```
public static void main(String[] args) {
    RemoteFileClient remoteFileClient = new RemoteFileClient("127.0.0.1", 3000);
    remoteFileClient.setUpConnection();
    String fileContents =
        remoteFileClient.getFile("C:\\WINNT\\Temp\\RemoteFile.txt");
    remoteFileClient.tearDownConnection();
    System.out.println(fileContents);
}
```

The `main()` method instantiates a new `RemoteFileClient` (the client) with an IP address and port number for the host. Then, we tell the client to set up a connection to the host (more on this later). Next, we tell the client to get the contents of a specified file on the host. Finally, we tell the client to tear down its connection to the host. We print out the contents of the file to the console, just to prove everything worked as planned.

Setting up a connection

Here we implement the `setUpConnection()` method, which will set up our `Socket` and give us access to its streams:

```
public void setUpConnection() {
    try {
        Socket client = new Socket(hostIp, hostPort);
        socketReader = new BufferedReader(
                            new InputStreamReader(client.getInputStream()));
        socketWriter = new PrintWriter(client.getOutputStream());
    } catch (UnknownHostException e) {
        System.out.println("Error setting up socket connection: unknown host at " + hos
    } catch (IOException e) {
        System.out.println("Error setting up socket connection: " + e);
    }
}
```

The `setUpConnection()` method creates a `Socket` with the IP address and port number of the host:

```
Socket client = new Socket(hostIp, hostPort);
```

We wrap the `Socket`'s `InputStream` in a `BufferedReader` so that we can read lines from the stream. Then, we wrap the `Socket`'s `OutputStream` in a `PrintWriter` so that we can send our request for a file to the server:

```
socketReader = new BufferedReader(new InputStreamReader(client.getInputStream()));
socketWriter = new PrintWriter(client.getOutputStream());
```

Remember that our client and server simply pass bytes back and forth. Both the client and the server have to know what the other is going to be sending so that they can respond appropriately. In this case, the server knows that we'll be sending it a valid file path.

When you instantiate a `Socket`, an `UnknownHostException` may be thrown. We don't do anything special to handle it here, but we print some information out to the console to tell us what went wrong. Likewise, if a general `IOException` is thrown when we try to get the `InputStream` or `OutputStream` on a `Socket`, we print out some information to the console. This is our general approach in this tutorial. In production code, we would be a little more sophisticated.

Talking to the host

Here we implement the `getFile()` method, which will tell the server what file we want and receive the contents from the server when it sends the contents back:

```
public String getFile(String fileNameToGet) {
    StringBuffer fileLines = new StringBuffer();
    try {
        socketWriter.println(fileNameToGet);
        socketWriter.flush();
        String line = null;
        while ((line = socketReader.readLine()) != null)
            fileLines.append(line + "\n");
    } catch (IOException e) {
        System.out.println("Error reading from file: " + fileNameToGet);
    }
    return fileLines.toString();
}
```

A call to the `getFile()` method requires a valid file path `String`. It starts by creating the `StringBuffer` called `fileLines` for storing each of the lines that we read from the file on the server:

```
StringBuffer fileLines = new StringBuffer();
```

In the `try{}catch{}` block, we send our request to the host using the `PrintWriter` that was established during connection setup:

```
socketWriter.println(fileNameToGet);
socketWriter.flush();
```

Note that we `flush()` the `PrintWriter` here instead of closing it. This forces data to be sent to the server without closing the `Socket`.

Once we've written to the `Socket`, we are expecting some response. We have to wait for it on the `Socket`'s `InputStream`, which we do by calling `readLine()` on our `BufferedReader` in a `while` loop. We append each returned line to the `fileLines` `StringBuffer` (with a newline character to preserve the lines):

```
String line = null;
while ((line = socketReader.readLine()) != null)
    fileLines.append(line + "\n");
```

Tearing down a connection

Here we implement the `tearDownConnection()` method, which will "clean up" after we're done using our connection:

```
public void tearDownConnection() {
    try {
        socketWriter.close();
        socketReader.close();
    } catch (IOException e) {
        System.out.println("Error tearing down socket connection: " + e);
    }
}
```

The `tearDownConnection()` method simply closes the `BufferedReader` and `PrintWriter` we created on our `Socket`'s `InputStream` and `OutputStream`, respectively. Doing this closes the underlying streams that we acquired from the `Socket`, so we have to catch the possible `IOException`.

Wrapping up the client

Our class is done. Before we move on to the server end of things, let's review the steps to create and use a `Socket`:

1. Instantiate a `Socket` with the IP address and port of the machine you're connecting to (throws an `Exception` if there's a problem).
2. Get the streams on that `Socket` for reading and writing.
3. Wrap the streams in instances of `BufferedReader`/`PrintWriter`, if that makes things easier.
4. Read from and write to the `Socket`.
5. Close your open streams.

You can find the complete code listing for `RemoteFileClient` at Code listing for RemoteFileClient on page 33.

Creating the RemoteFileServer class

Here is the structure for the `RemoteFileServer` class:

```
import java.io.*;
import java.net.*;
public class RemoteFileServer {
    protected int listenPort = 3000;
    public static void main(String[] args) {
```

```
    }
    public void acceptConnections() {
    }
    public void handleConnection(Socket incomingConnection) {
    }
}
```

As with the client, we first import `java.net` and `java.io`. Next, we give our class an instance variable to hold the port to listen to for incoming connections. By default, this is port 3000.

Our class has a `main()` method and two other methods. We'll go into the details of these methods later. For now, just know that `acceptConnections()` will allow clients to connect to the server, and `handleConnection()` interacts with the client `Socket` to send the contents of the requested file to the client.

Implementing main()

Here we implement the `main()` method, which will create a `RemoteFileServer` and tell it to accept connections:

```
public static void main(String[] args) {
    RemoteFileServer server = new RemoteFileServer();
    server.acceptConnections();
}
```

The `main()` method on the server side is even simpler than on the client side. We instantiate a new `RemoteFileServer`, which will listen for incoming connection requests on the default listen port. Then we call `acceptConnections()` to tell the server to listen.

Accepting connections

Here we implement the `acceptConnections()` method, which will set up a `ServerSocket` and wait for connection requests:

```
public void acceptConnections() {
    try {
        ServerSocket server = new ServerSocket(listenPort);
        Socket incomingConnection = null;
        while (true) {
            incomingConnection = server.accept();
            handleConnection(incomingConnection);
        }
    } catch (BindException e) {
        System.out.println("Unable to bind to port " + listenPort);
    } catch (IOException e) {
        System.out.println("Unable to instantiate a ServerSocket on port: " + listenPor
    }
}
```

The `acceptConnections()` method creates a `ServerSocket` with the port number to listen to. We then tell the `ServerSocket` to start listening by calling `accept()` on it. The

`accept()` method blocks until a connection request comes in. At that point, `accept()` returns a new `Socket` bound to a randomly assigned port on the server, which is passed to `handleConnection()`. Notice that this accepting of connections is in an infinite loop. No shutdown supported here.

Whenever you create a `ServerSocket`, Java code may throw an error if it can't *bind* to the specified port (perhaps because something else already has control of that port). So we have to catch the possible `BindException` here. And just like on the client side, we have to catch an `IOException` that could be thrown when we try to accept connections on our `ServerSocket`. Note that you can set a timeout on the `accept()` call by calling `setSoTimeout()` with number of milliseconds to avoid a really long wait. Calling `setSoTimeout()` will cause `accept()` to throw an `IOException` after the specified elapsed time.

Handling connections

Here we implement the `handleConnection()` method, which will use streams on a connection to receive input and write output:

```
public void handleConnection(Socket incomingConnection) {
    try {
        OutputStream outputToSocket = incomingConnection.getOutputStream();
        InputStream inputFromSocket = incomingConnection.getInputStream();
        BufferedReader streamReader =
            new BufferedReader(new InputStreamReader(inputFromSocket));
        FileReader fileReader = new FileReader(new File(streamReader.readLine()));
        BufferedReader bufferedFileReader = new BufferedReader(fileReader);
        PrintWriter streamWriter =
            new PrintWriter(incomingConnection.getOutputStream());
        String line = null;
        while ((line = bufferedFileReader.readLine()) != null) {
            streamWriter.println(line);
        }
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (Exception e) {
        System.out.println("Error handling a client: " + e);
    }
}
```

As with the client, we get the streams associated with the `Socket` we just made, using `getOutputStream()` and `getInputStream()`. As on the client side, we wrap the `InputStream` in a `BufferedReader` and the `OutputStream` in a `PrintWriter`. On the server side, we need to add some code to read the target file and send the contents to the client line by line. Here's the important code:

```
        FileReader fileReader = new FileReader(new File(streamReader.readLine()));
        BufferedReader bufferedFileReader = new BufferedReader(fileReader);
        String line = null;
        while ((line = bufferedFileReader.readLine()) != null) {
             streamWriter.println(line);
        }
```

This code deserves some detailed explanation. Let's look at it bit by bit:

```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
```

First, we make use of our `BufferedReader` on the `Socket`'s `InputStream`. We should be getting a valid file path, so we construct a new `File` using that path name. We make a new `FileReader` to handle reading the file.

```
BufferedReader bufferedFileReader = new BufferedReader(fileReader);
```

Here we wrap our `FileReader` in a `BufferedReader` to let us read the file line by line.

Next, we call `readLine()` on our `BufferedReader`. This call will block until bytes come in. When we get some bytes, we put them in our local `line` variable, and then write them out to the client. When we're done reading and writing, we close the open streams.

Note that we closed `streamWriter` and `streamReader` after we were done reading from the `Socket`. You might ask why we didn't close `streamReader` immediately after reading in the file name. The reason is that when you do that, your client won't get any data. If you close the `streamReader` before you close `streamWriter`, you can write to the `Socket` all you want but no data will make it across the channel (it's closed).

Wrapping up the server

Before we move on to another, more practical example, let's review the steps to create and use a `ServerSocket`:

1. Instantiate a `ServerSocket` with a port on which you want it to listen for incoming client connections (throws an `Exception` if there's a problem).
2. Call `accept()` on the `ServerSocket` to block while waiting for connection.
3. Get the streams on that underlying `Socket` for reading and writing.
4. Wrap the streams as necessary to simplify your life.
5. Read from and write to the `Socket`.
6. Close your open streams (and remember, *never* close your Reader before your Writer).

You can find the complete code listing for `RemoteFileServer` at Code listing for RemoteFileServer on page 34.

Section 5. A multithreaded example

Introduction

The previous example gives you the basics, but that won't take you very far. If you stopped here, you could handle only one client at a time. The reason is that `handleConnection()` is a blocking method. Only when it has completed its dealings with the current connection can the server accept another client. Most of the time, you will want (and need) a multithreaded server.

There aren't too many changes you need to make to `RemoteFileServer` to begin handling multiple clients simultaneously. As a matter of fact, had we discussed backlogging earlier, we would have just one method to change, although we'll need to create something new to handle the incoming connections. We will show you here also how `ServerSocket` handles lots of clients waiting (backing up) to use our server. This example illustrates an inefficient use of threads, so be patient.

Accepting (too many?) connections

Here we implement the revised `acceptConnections()` method, which will create a `ServerSocket` that can handle a backlog of requests, and tell it to accept connections:

```
public void acceptConnections() {
        try {
        ServerSocket server = new ServerSocket(listenPort, 5);
        Socket incomingConnection = null;
        while (true) {
            incomingConnection = server.accept();
            handleConnection(incomingConnection);
        }
    } catch (BindException e) {
    System.out.println("Unable to bind to port " + listenPort);
    } catch (IOException e) {
    System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
    }
}
```

Our new server still needs to `acceptConnections()` so this code is virtually identical. The highlighted line indicates the one significant difference. For this multithreaded version, we now specify the maximum number of client requests that can backlog when instantiating the `ServerSocket`. If we don't specify the max number of client requests, the default value of 50 is assumed.

Here's how it works. Suppose we specify a backlog of 5 and that five clients request connections to our server. Our server will start processing the first connection, but it takes a long time to process that connection. Since our backlog is 5, we can have up to five requests in the queue at one time. We're processing one, so that means we can have five others waiting. That's a total of six either waiting or being processed. If a seventh client asks for a connection while our server is still busy accepting connection one (remember that 2-6 are still in queue), that seventh client will be refused. We will illustrate limiting the number of clients that can be connected simultaneously in our pooled server example.

Handling connections: Part 1

Here we'll talk about the structure of the `handleConnection()` method, which spawns a new `Thread` to handle each connection. We'll discuss this in two parts. We'll focus on the method itself in this panel, and then examine the structure of the `ConnectionHandler` helper class used by this method in the next panel.

```
public void handleConnection(Socket connectionToHandle) {
     new Thread(new ConnectionHandler(connectionToHandle)).start();
}
```

This method represents the big change to our `RemoteFileServer`. We still call `handleConnection()` after the server accepts a connection, but now we pass that `Socket` to an instance of `ConnectionHandler`, which is `Runnable`. We create a new `Thread` with our `ConnectionHandler` and start it up. The `ConnectionHandler`'s `run()` method contains the `Socket` reading/writing and `File` reading code that used to be in `handleConnection()` on `RemoteFileServer`.

Handling connections: Part 2

Here is the structure for the `ConnectionHandler` class:

```
import java.io.*;
import java.net.*;
public class ConnectionHandler implements Runnable{
   Socket socketToHandle;
   public ConnectionHandler(Socket aSocketToHandle) {
      socketToHandle = aSocketToHandle;
   }
   public void run() {
   }
}
```

This helper class is pretty simple. As with our other classes so far, we import `java.net` and `java.io`. The class has a single instance variable, `socketToHandle`, that holds the `Socket` handled by the instance.

The constructor for our class takes a Socket instance and assigns it to `socketToHandle`.

Notice that the class implements the `Runnable` interface. Classes that implement this interface must implement the `run()` method, which our class does. We'll go into the details of `run()` later. For now, just know that it will actually process the connection using code identical to what we saw before in our `RemoteFileServer` class.

Implementing run()

Here we implement the `run()` method, which will grab the streams on our connection, use them to read from and write to the connection, and close them when we are done:

```
    public void run() {
        try {
```

```
            PrintWriter streamWriter = new PrintWriter(socketToHandle.getOutputStream(
            BufferedReader streamReader =
                new BufferedReader(new InputStreamReader(socketToHandle.getInputStream
            String fileToRead = streamReader.readLine();
            BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead))
            String line = null;
            while ((line = fileReader.readLine()) != null)
                streamWriter.println(line);
            fileReader.close();
            streamWriter.close();
            streamReader.close();
        } catch (Exception e) {
            System.out.println("Error handling a client: " + e);
        }
    }
```

The `run()` method on `ConnectionHandler` does what `handleConnection()` on `RemoteFileServer` did. First, we wrap the `InputStream` and `OutputStream` in a `BufferedReader` and a `PrintWriter`, respectively (using `getOutputStream()` and `getInputStream()` on the `Socket`). Then we read the target file line by line with this code:

```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
        BufferedReader bufferedFileReader = new BufferedReader(fileReader);
        String line = null;
        while ((line = bufferedFileReader.readLine()) != null) {
             streamWriter.println(line);
        }
```

Remember that we should be getting a valid file path from the client, so we construct a new `File` using that path name, wrap it in a `FileReader` to handle reading the file, and then wrap that in a `BufferedReader` to let us read the file line by line. We call `readLine()` on our `BufferedReader` in a `while` loop until we have no more lines to read. Remember that the call to `readLine()` will block until bytes come in. When we get some bytes, we put them in our local `line` variable, and then write them out to the client. When we're done reading and writing, we close the open streams.

Wrapping up the multithreaded server

Our multithreaded server is done. Before we move on to the pooled example, let's review the steps to create and use a multithreaded version of the server:

1. Modify `acceptConnections()` to instantiate a `ServerSocket` with a default 50-connection backlog (or whatever specific number you want, greater than 1).

2. Modify `handleConnection()` on the `ServerSocket` to spawn a new `Thread` with an instance of `ConnectionHandler`.

3. Implement the `ConnectionHandler` class, borrowing code from the `handleConnection()` method on `RemoteFileServer`.

You can find the complete code listing for `MultithreadedRemoteFileServer` at Code listing for MultithreadedRemoteFileServer on page 35, and the complete code listing for `ConnectionHandler` at Code listing for ConnectionHandler on page 35.

Section 6. A pooled example

Introduction

The `MultithreadedServer` we've got now simply creates a new `ConnectionHandler` in a new `Thread` each time a client asks for a connection. That means we have potentially a bunch of `Threads` lying around. Creating a `Thread` isn't trivial in terms of system overhead, either. If performance becomes an issue (and don't assume it will until it does), being more efficient about handling our server would be a good thing. So, how do we manage the server side more efficiently? We can maintain a pool of incoming connections that a limited number of ConnectionHandlers will service. This design provides the following benefits:

* It limits the number of simultaneous connections allowed.
* We only have to start up `ConnectionHandler Threads` one time.

Fortunately, as with our multithreaded example, adding pooling to our code doesn't require an overhaul. In fact, the client side of the application isn't affected at all. On the server side, we create a set number of `ConnectionHandlers` when the server starts, place incoming connections into a pool and let the `ConnectionHandlers` take care of the rest. There are many possible tweaks to this design that we won't cover. For instance, we could refuse clients by limiting the number of connections we allow to build up in the pool.

Note: We will not cover `acceptConnections()` again. This method is exactly the same as in earlier examples. It loops forever calling `accept()` on a `ServerSocket` and passes the connection to `handleConnection()`.

Creating the PooledRemoteFileServer class

Here is the structure for the `PooledRemoteFileServer` class:

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PooledRemoteFileServer {
    protected int maxConnections;
    protected int listenPort;
    protected ServerSocket serverSocket;
    public PooledRemoteFileServer(int aListenPort, int maxConnections) {
        listenPort = aListenPort;
        this.maxConnections = maxConnections;
    }
    public static void main(String[] args) {
    }
    public void setUpHandlers() {
    }
    public void acceptConnections() {
    }
    protected void handleConnection(Socket incomingConnection) {
    }
}
```

Note the `import` statements that should be familiar by now. We give our class the following instance variables to hold:

* The maximum number of simultaneous active client connections our server can handle
* The port to listen to for incoming connections (we didn't assign a default value, but feel free to do that if you want)
* The `ServerSocket` that will accept client connection requests

The constructor for our class takes the port to listen to and the maximum number of connections.

Our class has a `main()` method and three other methods. We'll go into the details of these methods later. For now, just know that `setUpHandlers()` creates a number of `PooledConnectionHandler` instances equal to `maxConnections` and the other two methods are like what we've seen before: `acceptConnections()` listens on the `ServerSocket` for incoming client connections, and `handleConnection` actually handles each client connection once it's established.

Implementing main()

Here we implement the revised `main()` method, which will create a `PooledRemoteFileServer` that can handle a given number of client connections, and tell it to accept connections:

```
public static void main(String[] args) {
    PooledRemoteFileServer server = new PooledRemoteFileServer(3000, 3);
    server.setUpHandlers();
    server.acceptConnections();
}
```

Our `main()` method is straightforward. We instantiate a new `PooledRemoteFileServer`, which will set up three `PooledConnectionHandlers` by calling `setUpHandlers()`. Once the server is ready, we tell it to `acceptConnections()`.

Setting up the connection handlers

```
public void setUpHandlers() {
    for (int i = 0; i < maxConnections; i++) {
        PooledConnectionHandler currentHandler = new PooledConnectionHandler();
        new Thread(currentHandler, "Handler " + i).start();
    }
}
```

The `setUpHandlers()` method creates `maxConnections` worth of `PooledConnectionHandlers` (three) and fires them up in new `Threads`. Creating a `Thread` with an object that implements `Runnable` allows us to call `start()` on the `Thread` and expect `run()` to be called on the `Runnable`. In other words, our `PooledConnectionHandlers` will be waiting to handle incoming connections, each in its own `Thread`. We create only three `Threads` in our example, and this cannot change once the server is running.

Handling connections

Here we implement the revised `handleConnections()` method, which will delegate handling a connection to a `PooledConnectionHandler`:

```
protected void handleConnection(Socket connectionToHandle) {
PooledConnectionHandler.processRequest(connectionToHandle);
}
```

We now ask our `PooledConnectionHandlers` to process all incoming connections (`processRequest()` is a static method).

Here is the structure for the `PooledConnectionHandler` class:

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PooledConnectionHandler implements Runnable {
    protected Socket connection;
    protected static List pool = new LinkedList();
    public PooledConnectionHandler() {
    }
    public void handleConnection() {
    }
    public static void processRequest(Socket requestToHandle) {
    }
    public void run() {
    }
}
```

This helper class is very much like `ConnectionHandler`, but with a twist to handle connection pooling. The class has two single instance variables:

* `connection`, the `Socket` that is currently being handled
* A static `LinkedList` called `pool` that holds the connections that need to be handled

Filling the connection pool

Here we implement the `processRequest()` method on our `PooledConnectionHandler`, which will add incoming requests to the pool and tell other objects waiting on the pool that it now has some contents:

```
public static void processRequest(Socket requestToHandle) {
    synchronized (pool) {
        pool.add(pool.size(), requestToHandle);
        pool.notifyAll();
    }
}
```

This method requires some background on how the Java keyword `synchronized` works. We will attempt a short lesson on threading.

First, some definitions:

* **Atomic method.** Methods (or blocks of code) that cannot be interrupted mid-execution
* **Mutex lock.** A single "lock" that must be obtained by a client wishing to execute an atomic method

So, when object A wants to use `synchronized` method `doSomething()` on object B, object A must first attempt to acquire the mutex from object B. Yes, this means that when object A has the mutex, no other object may call *any* other `synchronized` method on object B.

A `synchronized` block is a slightly different animal. You can synchronize a block on any object, not just the object that has the block in one of its methods. In our example, our `processRequest()` method contains a block `synchronized` on the `pool` object (remember it's a `LinkedList` that holds the pool of connections to be handled). The reason we do this is to ensure that nobody else can modify the connection pool at the same time we are.

Now that we've guaranteed that we're the only ones wading in the pool, we can add the incoming `Socket` to the end of our `LinkedList`. Once we've added the new connection, we notify other `Threads` waiting to access the pool that it's now available, using this code:

```
pool.notifyAll();
```

All subclasses of `Object` inherit the `notifyAll()` method. This method, in conjunction with the `wait()` method that we'll discuss in the next panel, allows one `Thread` to let another `Thread` know that some condition has been met. That means that the second `Thread` must have been waiting for that condition to be satisfied.

Getting connections from the pool

Here we implement the revised `run()` method on `PooledConnectionHandler`, which will wait on the connection pool and handle the connection once the pool has one:

```
public void run() {
        while (true) {
             synchronized (pool) {
                  while (pool.isEmpty()) {
                       try {
                            pool.wait();
                       } catch (InterruptedException e) {
                            return;
                       }
                   }
                   connection = (Socket) pool.remove(0);
             }
             handleConnection();
        }
}
```

Recall from the previous panel that a `Thread` is waiting to be notified that a condition on the connection pool has been satisfied. In our example, remember that we have three

PooledConnectionHandlers waiting to use connections in the pool. Each of these PooledConnectionHandlers is running in its own Thread and is blocked on the call to pool.wait(). When our processRequest() method called notifyAll() on the connection pool, all of our waiting PooledConnectionHandlers were notified that the pool was available. Each one then continues past the call to pool.wait(), and rechecks the while(pool.isEmpty()) loop condition. The pool will be empty for all but one handler, so all but one handler will block again on the call to pool.wait(). The one that encounters a non-empty pool will break out of the while(pool.isEmpty()) loop and will grab the first connection from the pool:

```
connection = (Socket) pool.remove(0);
```

Once it has a connection to use, it calls handleConnection() to handle it.

In our example, the pool probably won't ever have more than one connection in it, simply because things execute so fast. If there were more than one connection in the pool, then the other handlers wouldn't have to wait for new connections to be added to the pool. When they checked the pool.isEmpty() condition, it would fail, and they would proceed to grab a connection from the pool and handle it.

One other thing to note. How is the processRequest() method able to put connections in the pool when the run() method has a mutex lock on the pool? The answer is that the call to wait() on the pool releases the lock, and then grabs it again right before it returns. This allows other code synchronized on the pool object to acquire the lock.

Handling connections: One more time

Here we implement the revised handleConnection() method, which will grab the streams on a connection, use them, and clean them up when finished:

```
public void handleConnection() {
    try {
        PrintWriter streamWriter = new PrintWriter(connection.getOutputStream());
        BufferedReader streamReader =
            new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String fileToRead = streamReader.readLine();
        BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));
        String line = null;
        while ((line = fileReader.readLine()) != null)
            streamWriter.println(line);
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (FileNotFoundException e) {
        System.out.println("Could not find requested file on the server.");
    } catch (IOException e) {
        System.out.println("Error handling a client: " + e);
    }
}
```

Unlike in our multithreaded server, our PooledConnectionHandler has a handleConnection() method. The code within this method is exactly the same as the code in the run() method on our non-pooled ConnectionHandler. First, we wrap the

`OutputStream` and `InputStream` in a `PrintWriter` and a `BufferedReader`, respectively (using `getOutputStream()` and `getInputStream()` on the `Socket`). Then we read the target file line by line, just as we did in the multithreaded example. Again, when we get some bytes, we put them in our local `line` variable, and then write them out to the client. When we're done reading and writing, we close our `FileReader` and the open streams.

Wrapping up the pooled server

Our pooled server is done. Let's review the steps to create and use a pooled version of the server:

1. Create a new kind of connection handler (we called it `PooledConnectionHandler`) to handle connections in a pool.

2. Modify the server to create and use a set of `PooledConnectionHandlers`.

You can find the complete code listing for `PooledRemoteFileServer` at Code listing for PooledRemoteFileServer on page 36, and the complete code listing for `PooledConnectionHandler` at Code listing for PooledConnectionHandler on page 37.

Section 7. Sockets in real life

Introduction

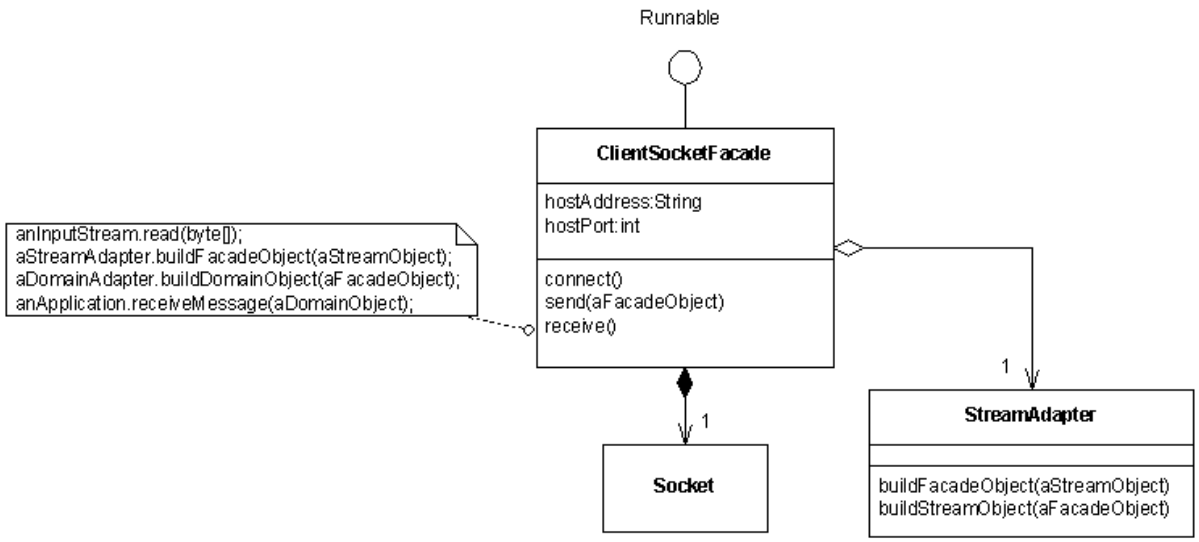
The examples we've talked about so far cover the mechanics of sockets in Java programming, but how would you use them on something "real?" Such a simple use of sockets, even with multithreading and pooling, would not be appropriate in most applications. Instead, it would probably be smart to use sockets within other classes that model your problem domain.

We did this recently in porting an application from a mainframe/SNA environment to a TCP/IP environment. The application's job is to facilitate communication between a retail outlet (such as a hardware store) and financial institutions. Our application is the middleman. As such, it needs to communicate with the retail outlet on one side and the financial outlet on the other. We had to handle a client talking to a server via sockets, and we had to translate our domain objects into socket-ready stuff for transmission.

We can't cover all the detail of this application in this tutorial, but let us take you on a tour of some of the high points. You can extrapolate from here to your own problem domain.

The client side

On the client side, the key players in our system were `Socket`, `ClientSocketFacade`, and `StreamAdapter`. The UML is shown in the following diagram:



We created a `ClientSocketFacade`, which is `Runnable` and owns an instance of `Socket`. Our application can instantiate a `ClientSocketFacade` with a particular host IP address and port number, and run it in a new `Thread`. The `run()` method on `ClientSocketFacade` calls `connect()`, which lazily initializes a `Socket`. With `Socket` instance in hand, our `ClientSocketFacade` calls `receive()` on itself, which blocks until the server sends some data over the `Socket`. Whenever the server sends some data, our `ClientSocketFacade` will wake up and handle the incoming data. Sending data is just as direct. Our application can simply tell its `ClientSocketFacade` to send data to its server by

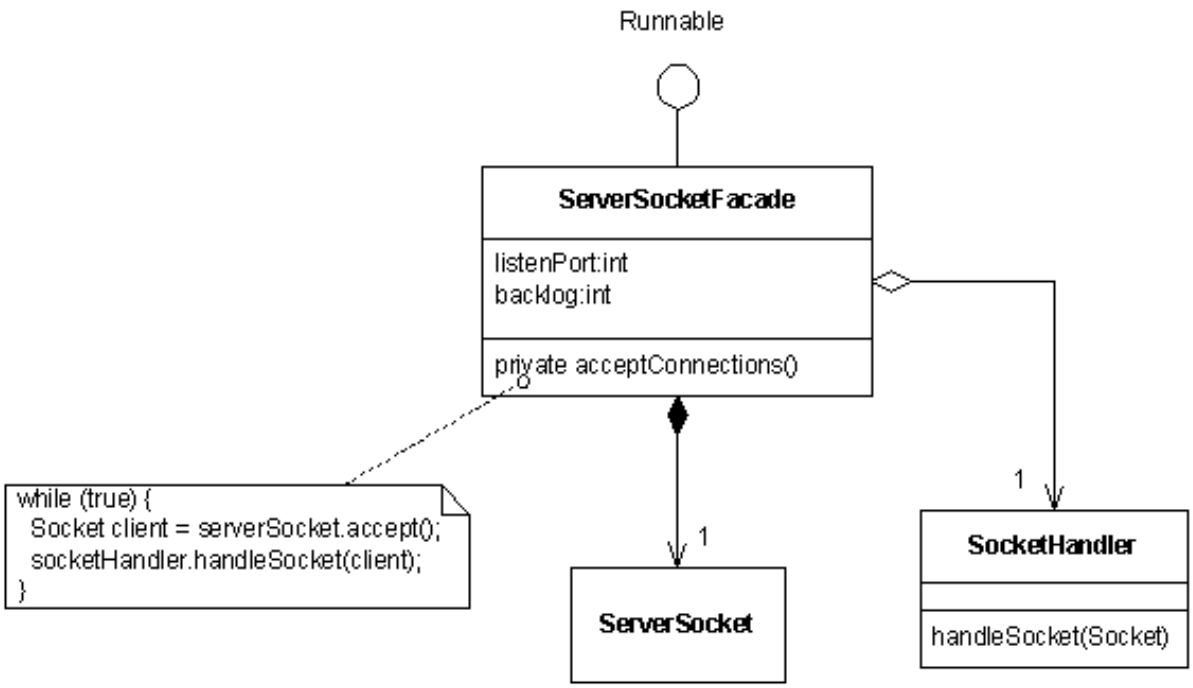
calling the `send()` method with a `StreamObject`.

The only piece missing from the discussion above is `StreamAdapter`. When an application tells the `ClientSocketFacade` to send data, the Facade delegates the operation to an instance of `StreamAdapter`. The `ClientSocketFacade` delegates receiving data to the same instance of `StreamAdapter`. A `StreamAdapter` handles the final formatting of messages to put on the `Socket`'s `OutputStream`, and reverses the process for messages coming in on the `Socket`'s `InputStream`.

For example, perhaps your server needs to know the number of bytes in the message being sent. `StreamAdapter` could handle computing and prepending the length to the message before sending it. When the server receives it, the same `StreamAdapter` could handle stripping off the length and reading the correct number of bytes for building a `StreamReadyObject`.

The server side

The picture is similar on the server side:



We wrapped our `ServerSocket` in a `ServerSocketFacade`, which is `Runnable` and owns an instance of a `ServerSocket`. Our applications can instantiate a `ServerSocketFacade` with a particular server-side port to listen to and a maximum number of client connections allowed (the default is 50). The application then runs the Facade in a new `Thread` to hide the `ServerSocket` interaction details.

The `run()` method on `ServerSocketFacade` calls `acceptConnections()`, which makes a new `ServerSocket` and calls `accept()` on it to block until a client requests a connection. Each time that happens, our `ServerSocketFacade` wakes up and hands the new `Socket` returned by `accept()` to an instance of `SocketHandler` by calling

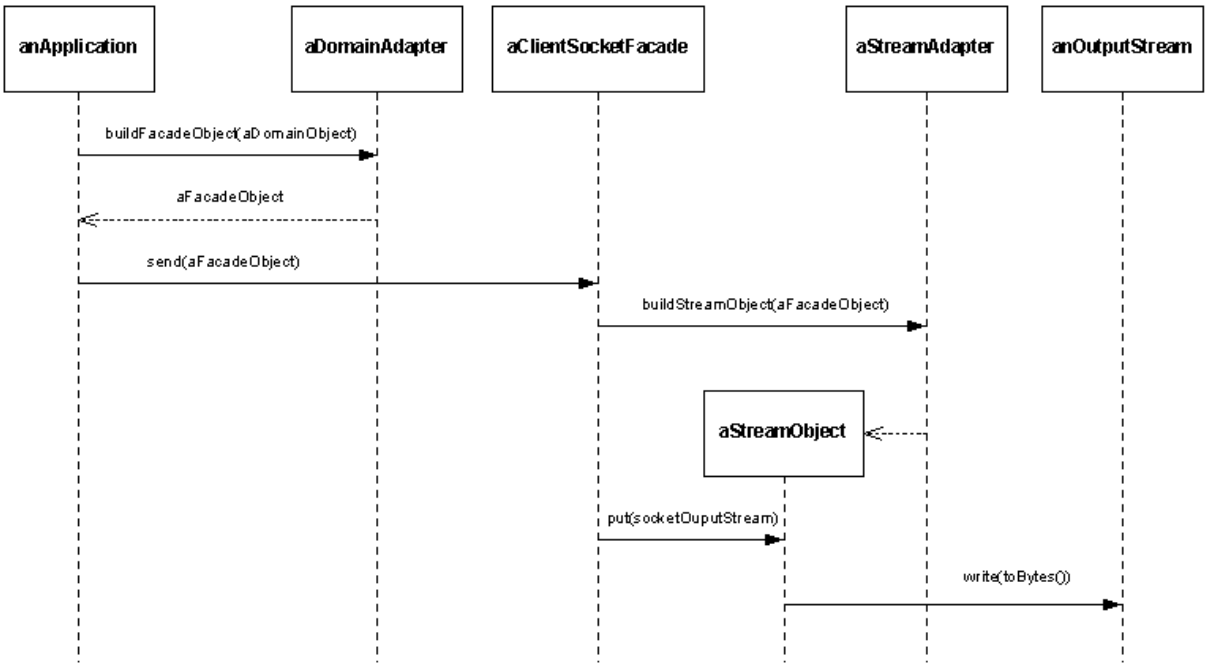
`handleSocket()`. The `SocketHandler` does what it needs to do in order to handle the new channel from client to server.

The business logic

Once we had these `Socket` Facades in place, it became much easier to implement the business logic of our application. Our application used an instance of `ClientSocketFacade` to send data over the `Socket` to the server and to get responses back. The application was responsible for handling conversion of our domain objects into formats understood by `ClientSocketFacade` and for building domain objects from responses.

Sending messages to the server

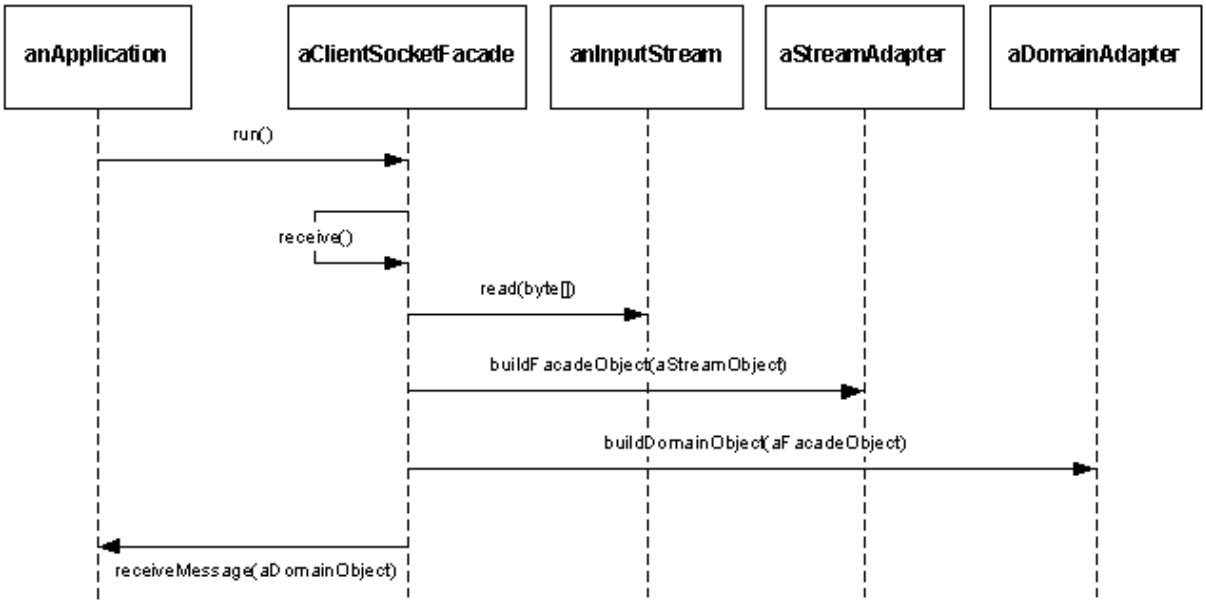
The following diagram shows the UML interaction diagram for sending a message in our application:



For simplicity's sake, we didn't show the piece of the interaction where `aClientSocketFacade` asks its `Socket` instance for its `OutputStream` (using the `getOutputStream()` method). Once we had a reference to that `OutputStream`, we simply interacted with it as shown in the diagram. Notice that our `ClientSocketFacade` hid the low-level details of socket interaction from our application. Our application interacted with `aClientSocketFacade`, not with any of the lower-level classes that facilitate putting bytes on `Socket OutputStreams`.

Receiving messages from the server

The following diagram shows the UML interaction diagram for receiving a message in our application:



Notice that our application runs `aClientSocketFacade` in a `Thread`. When `aClientSocketFacade` starts up, it tells itself to `receive()` on its `Socket` instance's `InputStream`. The `receive()` method calls `read(byte[])` on the `InputStream` itself. The `read([])` method blocks until it receives data, and puts the bytes received on the `InputStream` into a byte array. When data comes in, `aClientSocketFacade` uses `aStreamAdapter` and `aDomainAdapter` to construct (ultimately) a domain object that our application can use. Then it hands that domain object back to the application. Again, our `ClientSocketFacade` hides the lower-level details from the application, simplifying the Application Layer.

Section 8. Summary

Wrapup

The Java language simplifies using sockets in your applications. The basics are really the `Socket` and `ServerSocket` classes in the `java.net` package. Once you understand what's going on behind the scenes, these classes are easy to use. Using sockets in real life is simply a matter of using good OO design principles to preserve encapsulation within the various layers within your application. We showed you a few classes that can help. The structure of these classes hides the low-level details of `Socket` interaction from our application -- it can just use pluggable `ClientSocketFacades` and `ServerSocketFacades`. You still have to manage the somewhat messy byte details somewhere (within the Facades), but you can do it once and be done with it. Better still, you can reuse these lower-level helper classes on future projects.

Resources

* Download the *source code* for this article.

* "*Thinking in Java, 2nd Edition*" (Prentice Hall, 2000) by Bruce Eckel provides an excellent approach for learning Java inside and out.

* Sun has a good tutorial on *Sockets*. Just follow the "All About Sockets" link.

* We used VisualAge for Java, version 3.5 to develop the code in this tutorial. Download your own copy of *VisualAge for Java* (now in release 4) or, if you already use VAJ, check out the *VisualAge Developer Domain* for a variety of technical assistance.

* Now that you're up to speed with sockets programming with Java, this article on the Visual Age for Java Developer Domain will teach you to *set up access to sockets through the company firewall*.

* Allen Holub's *Java Toolbox column* (on *JavaWorld*) provides an excellent series on Java Threads that is well worth reading. Start the series with "*A Java programmer's guide to threading architectures* ." One particularly good article, "*Threads in an object-oriented world, thread pools, implementing socket 'accept' loops,* " goes into rather deep detail about `Thread` pooling. We didn't go into quite so much detail in this tutorial, and we made our `PooledRemoteFileServer` and `PooledConnectionHandler` a little easier to follow, but the strategies Allen talks about would fit nicely. In fact, his treatment of `ServerSocket` via a Java implementation of a callback mechanism that supports a multi-purpose, configurable server is powerful.

* For technical assistance with multithreading in your Java applications, visit the *Multithreaded Java programming discussion forum* on developerWorks, moderated by Java threading expert Brian Goetz.

* Siva Visveswaran explains connection pooling in detail in *"Connection pools"* (developerWorks, October 2000).

Your feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Section 9. Appendix

Code listing for URLClient

```
import java.io.*;
import java.net.*;
public class URLClient {
    protected HttpURLConnection connection;
    public String getDocumentAt(String urlString) {
        StringBuffer document = new StringBuffer();
        try {
            URL url = new URL(urlString);
            URLConnection conn = url.openConnection();
            BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getIn
            String line = null;
            while ((line = reader.readLine()) != null)
                document.append(line + "\n");
            reader.close();
        } catch (MalformedURLException e) {
            System.out.println("Unable to connect to URL: " + urlString);
        } catch (IOException e) {
            System.out.println("IOException when connecting to URL: " + urlString);
        }
        return document.toString();
    }
    public static void main(String[] args) {
        URLClient client = new URLClient();
        String yahoo = client.getDocumentAt("http://www.yahoo.com");
        System.out.println(yahoo);
    }
}
```

Code listing for RemoteFileClient

```
import java.io.*;
import java.net.*;
public class RemoteFileClient {
    protected BufferedReader socketReader;
    protected PrintWriter socketWriter;
    protected String hostIp;
    protected int hostPort;
    public RemoteFileClient(String aHostIp, int aHostPort) {
        hostIp = aHostIp;
        hostPort = aHostPort;
    }
    public String getFile(String fileNameToGet) {
        StringBuffer fileLines = new StringBuffer();
        try {
            socketWriter.println(fileNameToGet);
            socketWriter.flush();
            String line = null;
            while ((line = socketReader.readLine()) != null)
                fileLines.append(line + "\n");
        } catch (IOException e) {
            System.out.println("Error reading from file: " + fileNameToGet);
        }
        return fileLines.toString();
    }
    public static void main(String[] args) {
```

```
        RemoteFileClient remoteFileClient = new RemoteFileClient("127.0.0.1", 3000);
        remoteFileClient.setUpConnection();
        String fileContents = remoteFileClient.getFile("C:\\WINNT\\Temp\\RemoteFile.txt
        remoteFileClient.tearDownConnection();
        System.out.println(fileContents);
    }
    public void setUpConnection() {
        try {
            Socket client = new Socket(hostIp, hostPort);
            socketReader = new BufferedReader(new InputStreamReader(client.getInputStre
            socketWriter = new PrintWriter(client.getOutputStream());
        } catch (UnknownHostException e) {
            System.out.println("Error setting up socket connection: unknown host at " +
        } catch (IOException e) {
            System.out.println("Error setting up socket connection: " + e);
        }
    }
    public void tearDownConnection() {
        try {
            socketWriter.close();
            socketReader.close();
        } catch (IOException e) {
            System.out.println("Error tearing down socket connection: " + e);
        }
    }
}
```

Code listing for RemoteFileServer

```
import java.io.*;
import java.net.*;
public class RemoteFileServer {
    int listenPort;
    public RemoteFileServer(int aListenPort) {
        listenPort = aListenPort;
    }
    public void acceptConnections() {
        try {
            ServerSocket server = new ServerSocket(listenPort);
            Socket incomingConnection = null;
            while (true) {
                incomingConnection = server.accept();
                handleConnection(incomingConnection);
            }
        } catch (BindException e) {
            System.out.println("Unable to bind to port " + listenPort);
        } catch (IOException e) {
            System.out.println("Unable to instantiate a ServerSocket on port: " + liste
        }
    }
    public void handleConnection(Socket incomingConnection) {
        try {
            OutputStream outputToSocket = incomingConnection.getOutputStream();
            InputStream inputFromSocket = incomingConnection.getInputStream();
            BufferedReader streamReader = new BufferedReader(new InputStreamReader(inpu
            FileReader fileReader = new FileReader(new File(streamReader.readLine()));
            BufferedReader bufferedFileReader = new BufferedReader(fileReader);
            PrintWriter streamWriter = new PrintWriter(incomingConnection.getOutputStre
            String line = null;
            while ((line = bufferedFileReader.readLine()) != null) {
                streamWriter.println(line);
```

```
            }
            fileReader.close();
            streamWriter.close();
            streamReader.close();
        } catch (Exception e) {
            System.out.println("Error handling a client: " + e);
        }
    }
    public static void main(String[] args) {
        RemoteFileServer server = new RemoteFileServer(3000);
        server.acceptConnections();
    }
}
```

Code listing for MultithreadedRemoteFileServer

```
import java.io.*;
import java.net.*;
public class MultithreadedRemoteFileServer {
    protected int listenPort;
    public MultithreadedRemoteFileServer(int aListenPort) {
        listenPort = aListenPort;
    }
    public void acceptConnections() {
        try {
            ServerSocket server = new ServerSocket(listenPort, 5);
            Socket incomingConnection = null;
            while (true) {
                incomingConnection = server.accept();
                handleConnection(incomingConnection);
            }
        } catch (BindException e) {
            System.out.println("Unable to bind to port " + listenPort);
        } catch (IOException e) {
            System.out.println("Unable to instantiate a ServerSocket on port: " + liste
        }
    }
    public void handleConnection(Socket connectionToHandle) {
        new Thread(new ConnectionHandler(connectionToHandle)).start();
    }
    public static void main(String[] args) {
        MultithreadedRemoteFileServer server = new MultithreadedRemoteFileServer(3000)
        server.acceptConnections();
    }
}
```

Code listing for ConnectionHandler

```
import java.io.*;
import java.net.*;
public class ConnectionHandler implements Runnable {
    protected Socket socketToHandle;
    public ConnectionHandler(Socket aSocketToHandle) {
        socketToHandle = aSocketToHandle;
    }
    public void run() {
        try {
            PrintWriter streamWriter = new PrintWriter(socketToHandle.getOutputStream(
```

```
            BufferedReader streamReader = new BufferedReader(new InputStreamReader(sock
            String fileToRead = streamReader.readLine();
            BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));
            String line = null;
            while ((line = fileReader.readLine()) != null)
                streamWriter.println(line);
            fileReader.close();
            streamWriter.close();
            streamReader.close();
        } catch (Exception e) {
            System.out.println("Error handling a client: " + e);
        }
    }
}
```

Code listing for PooledRemoteFileServer

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PooledRemoteFileServer {
    protected int maxConnections;
    protected int listenPort;
    protected ServerSocket serverSocket;
    public PooledRemoteFileServer(int aListenPort, int maxConnections) {
        listenPort = aListenPort;
        this.maxConnections = maxConnections;
    }
    public void acceptConnections() {
        try {
            ServerSocket server = new ServerSocket(listenPort, 5);
            Socket incomingConnection = null;
            while (true) {
                incomingConnection = server.accept();
                handleConnection(incomingConnection);
            }
        } catch (BindException e) {
            System.out.println("Unable to bind to port " + listenPort);
        } catch (IOException e) {
            System.out.println("Unable to instantiate a ServerSocket on port: " + liste
        }
    }
    protected void handleConnection(Socket connectionToHandle) {
        PooledConnectionHandler.processRequest(connectionToHandle);
    }
    public static void main(String[] args) {
        PooledRemoteFileServer server = new PooledRemoteFileServer(3000, 3);
        server.setUpHandlers();
        server.acceptConnections();
    }
    public void setUpHandlers() {
        for (int i = 0; i < maxConnections; i++) {
            PooledConnectionHandler currentHandler = new PooledConnectionHandler();
            new Thread(currentHandler, "Handler " + i).start();
        }
    }
}
```

Code listing for PooledConnectionHandler

```
import java.io.*;
import java.net.*;
import java.util.*;
public class PooledConnectionHandler implements Runnable {
    protected Socket connection;
    protected static List pool = new LinkedList();
    public PooledConnectionHandler() {
    }
    public void handleConnection() {
        try {
            PrintWriter streamWriter = new PrintWriter(connection.getOutputStream());
            BufferedReader streamReader = new BufferedReader(new InputStreamReader(conn
            String fileToRead = streamReader.readLine();
            BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead))
            String line = null;
            while ((line = fileReader.readLine()) != null)
                streamWriter.println(line);
            fileReader.close();
            streamWriter.close();
            streamReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("Could not find requested file on the server.");
        } catch (IOException e) {
            System.out.println("Error handling a client: " + e);
        }
    }
    public static void processRequest(Socket requestToHandle) {
        synchronized (pool) {
            pool.add(pool.size(), requestToHandle);
            pool.notifyAll();
        }
    }
    public void run() {
        while (true) {
            synchronized (pool) {
                while (pool.isEmpty()) {
                    try {
                        pool.wait();
                    } catch (InterruptedException e) {
                        return;
                    }
                }
                connection = (Socket) pool.remove(0);
            }
            handleConnection();
        }
    }
}
```

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.