© 2012 Marty Hall

Network Programming: Servers

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/java.html

Customized Java EE Training: http://courses.coreservlets.com/
Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

Agenda

- **Steps for creating a server**
  1. Create a ServerSocket object
  2. Create a Socket object from ServerSocket
  3. Create an input stream
  4. Create an output stream
  5. Do I/O with input and output streams
  6. Close the socket
- **A generic network server**
  – Single threaded
  – Multithreaded
- **Accepting connections from browsers**
- **A simple HTTP server**

Steps for Implementing a Server

1. **Create a ServerSocket object**

```
ServerSocket listenSocket =
  new ServerSocket(portNumber);
```

2. **Create a Socket object from ServerSocket**

```
while(someCondition) {
  Socket server = listenSocket.accept();
  doSomethingWith(server);
}
```

- Note that it is quite common to have doSomethingWith spin off a separate thread

3. **Create an input stream to read client input**

```
BufferedReader in =
 new BufferedReader
   (new InputStreamReader(server.getInputStream()));
```

Steps for Implementing a Server

4. **Create an output stream that can be used to send info back to the client.**

```
// Last arg of true means autoflush stream
// when println is called
PrintWriter out =
  new PrintWriter(server.getOutputStream(), true)
```

5. **Do I/O with input and output Streams**
   - Most common input: readLine
   - Most common output: println
   - Again you can use ObjectInputStream and ObjectOutputStream for Java-to-Java communication
6. **Close the socket when done**

```
server.close();
```

   - This closes the associated input and output streams.

A Generic Single-Threaded Network Server

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

/** A starting point for network servers. */

public abstract class NetworkServer {
  private int port;

  /** Build a server on specified port. It will continue to
   *  accept connections, passing each to handleConnection until
   *  the server is killed (e.g., Control-C in the startup window)
   *  or System.exit() from handleConnection of elsewhere
   *  in the Java code).
   */

  public NetworkServer(int port) {
    this.port = port;
  }
```

A Generic Network Server (Continued)

```
/** Monitor a port for connections. Each time one
 *  is established, pass resulting Socket to
 *  handleConnection.
 */

public void listen() {
  try {
    ServerSocket listener = new ServerSocket(port);
    Socket socket;
    while(true) {  // Run until killed
      socket = listener.accept();
      handleConnection(socket);
    }
  } catch (IOException ioe) {
    System.out.println("IOException: " + ioe);
    ioe.printStackTrace();
  }
}
```

A Generic Network Server (Continued)

```
  /** This is the method that provides the behavior to the
   *  server, since it determines what is done with the
   *  resulting socket. <B>Override this method in servers
   *  you write.</B>
   */

  protected abstract void handleConnection(Socket socket)
      throws IOException;

  /** Gets port on which server is listening. */

  public int getPort() {
    return(port);
  }
}
```

Using Network Server

```
public class NetworkServerTest extends NetworkServer {
  public NetworkServerTest(int port) {
    super(port);
  }

  protected void handleConnection(Socket socket)
      throws IOException{
    PrintWriter out = SocketUtil.getWriter(socket);
    BufferedReader in = SocketUtil.getReader(socket);
    System.out.printf
      ("Generic Server: got connection from %s%n" +
       "with first line '%s'.%n",
       socket.getInetAddress().getHostName(),
       in.readLine());
    out.println("Generic Server");
    socket.close();
  }
```

Using Network Server (Continued)

```
  public static void main(String[] args) {
    int port = 8088;
    if (args.length > 0) {
      port = Integer.parseInt(args[0]);
    }
    NetworkServerTest tester =
      new NetworkServerTest(port);
    tester.listen();
  }
}
```

Network Server: Results

- **Accepting a Connection from a WWW Browser**
  - Suppose the above test program is started up on port 8088 of `server.com`:

```
server> java NetworkServerTest
```

  - Then, a standard Web browser on `client.com` requests `http://server.com:8088/foo/`, yielding the following back on `server.com`:

```
Generic Network Server:
got connection from client.com
with first line 'GET /foo/ HTTP/1.0'
```

Template for a Multithreaded Network Server

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

public class MultithreadedServer {
  private int port;

  public MultithreadedServer(int port) {
    this.port = port;
  }

  public int getPort() {
    return(port);
  }
```

MultithreadedServer.java (Continued)

```
public void listen() {
   int poolSize =
     50 * Runtime.getRuntime().availableProcessors();
   ExecutorService tasks =
     Executors.newFixedThreadPool(poolSize);
   try {
     ServerSocket listener = new ServerSocket(port);
     Socket socket;
     while(true) {  // Run until killed
       socket = listener.accept();
       tasks.execute(new ConnectionHandler(socket));
     }
   } catch (IOException ioe) {
     System.err.println("IOException: " + ioe);
     ioe.printStackTrace();
   }
 }}
```

The upcoming EchoServer will apply this template to making an HTTP server.

ConnectionHandler.java

```
public class ConnectionHandler implements Runnable {
  private Socket socket;

  public ConnectionHandler(Socket socket) {
    this.socket = socket;
  }

  public void run() {
    try {
      handleConnection(socket);
    } catch(IOException ioe) {
      System.err.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }

  public void handleConnection(Socket socket)
      throws IOException{
    // Do something with socket
  }
}
```

HTTP Requests and Responses

- **Request**

```
GET /~gates/ HTTP/1.1
Host: www.mainhost.com
Connection: close
Header3: …
…
HeaderN: …
```

Blank Line

  - All request headers are optional except for `Host` (required only for HTTP/1.1)
  - If you send `HEAD` instead of `GET`, the server returns the same HTTP headers, but no document

- **Response**

```
HTTP/1.0 200 OK
Content-Type: text/html
Header2: …
…
HeaderN: …
```

Blank Line

```
<!DOCTYPE …>
<HTML>
…
</HTML>
```

  - All response headers are optional except for `Content-Type`

A Simple HTTP Server

- **Idea**
  1. Read lines sent by the browser, storing them in a List
     - Use readLine a line at a time until an empty line
       - Exception: with POST requests you have to read extra line
  2. Send an HTTP response line (e.g. "HTTP/1.1 200 OK")
  3. Send a Content-Type line then a blank line
     - This indicates the file type being returned (HTML in this case)
  4. Send an HTML file showing the lines that were sent
     - Put the input in a PRE section inside the BODY
  5. Close the connection

EchoServer.java

```
/** A simple HTTP server that generates a Web page
 *  showing all of the data that it received from
 *  the Web client (usually a browser). */

public class EchoServer {
  private int port;

  public EchoServer(int port) {
    this.port = port;
  }

  public static void main(String[] args) {
    int port = 8088;
    if (args.length > 0) {
      try {
        port = Integer.parseInt(args[0]);
      } catch(NumberFormatException nfe) {}
    }
    EchoServer server = new EchoServer(port);
    server.listen();
  }
```

EchoServer.java (Continued)

```
public void listen() {
    int poolSize =
      50 * Runtime.getRuntime().availableProcessors();
    ExecutorService tasks =
      Executors.newFixedThreadPool(poolSize);
    try {
      ServerSocket listener = new ServerSocket(port);
      Socket socket;
      while(true) {  // Run until killed
        socket = listener.accept();
        tasks.execute(new EchoHandler(socket));
      }
    } catch (IOException ioe) {
      System.out.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }
```

EchoHandler.java

```
public class EchoHandler implements Runnable {
  private Socket socket;

  public EchoHandler(Socket socket) {
    this.socket = socket;
  }

  public void run() {
    try {
      handleConnection(socket);
    } catch(IOException ioe) {
      System.err.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }
```

EchoHandler.java (Continued)

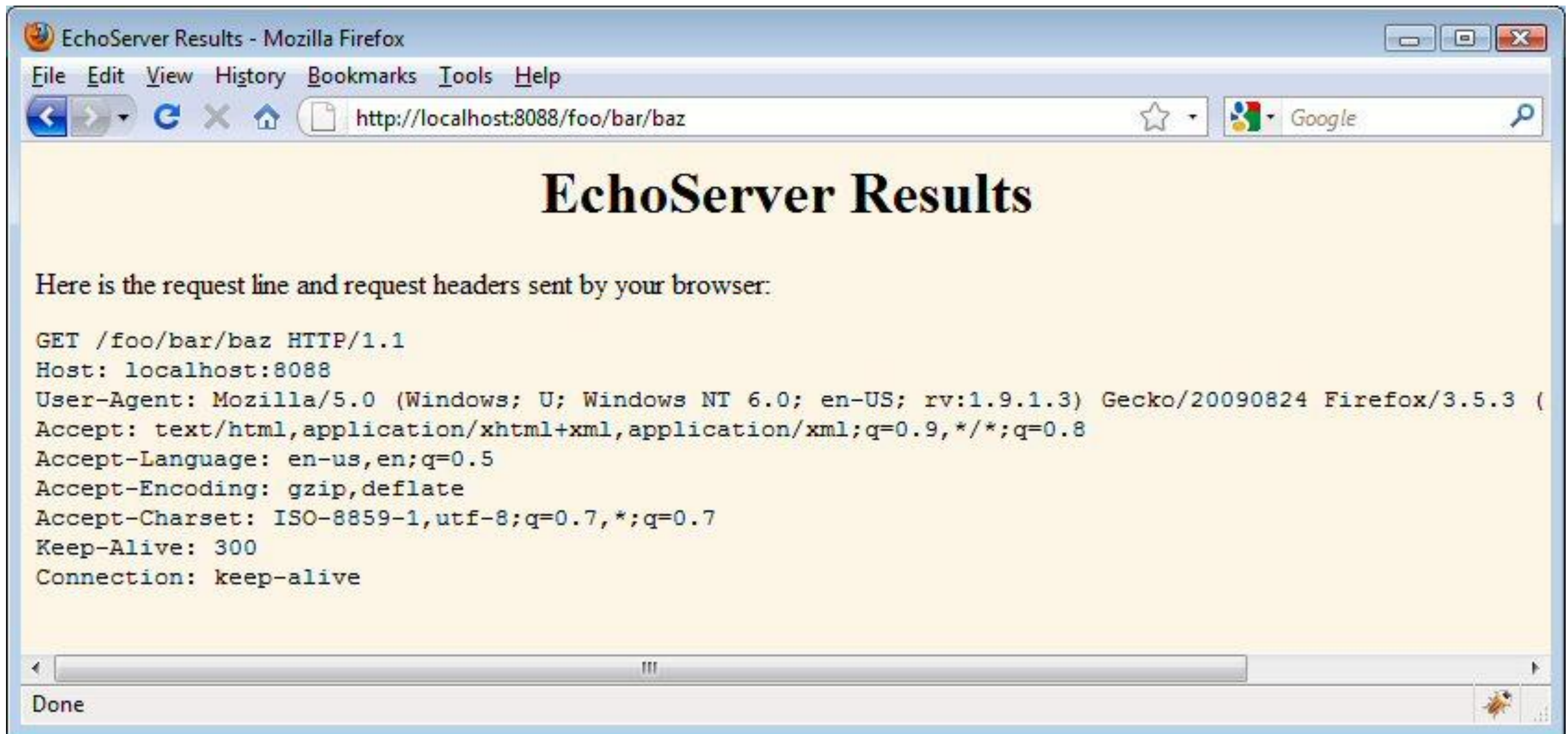
```
public void handleConnection(Socket socket)
     throws IOException{
   PrintWriter out = SocketUtil.getWriter(socket);
   BufferedReader in = SocketUtil.getReader(socket);
   List<String> inputLines = new ArrayList<String>();
   String line;
   while((line = in.readLine()) != null) {
     inputLines.add(line);
     if (line.length() == 0) { // Blank line.
       if (usingPost(inputLines)) { // One more line if POST
         …
       }
       break;
     }
   }
   printHeader(out);
   for (String inputLine: inputLines) {
     out.println(inputLine);
   }
   printTrailer(out);
   socket.close();
 }
```

EchoHandler.java (Continued)

```
private void printHeader(PrintWriter out) {
  String serverName = "EchoServer";
  out.println
    ("HTTP/1.1 200 OK\r\n" +
     "Server: " + serverName + "\r\n" +
     "Content-Type: text/html\r\n" +
     "\r\n" +
     "<!DOCTYPE HTML PUBLIC " +
       "\"-//W3C//DTD HTML 4.0 Transitional//EN\">\n" +
     …
     "<PRE>");
}

private void printTrailer(PrintWriter out) {
  out.println
    ("</PRE></BODY></HTML>\n");
}
```

EchoServer in Action



Summary

- **Create a ServerSocket; specify port number**
  - Call accept to wait for a client connection
  - accept returns a Socket object (just as in last lecture)
- **Browser requests:**
  - GET, POST, or HEAD line
  - 0 or more request headers
  - blank line
  - One additional line (query data) for POST requests only
- **HTTP server response:**
  - Status line (HTTP/1.0 200 OK),
  - Content-Type (and, optionally, other response headers)
  - Blank line
  - Document
- **For improved performance**
  - Make multithreaded task queue to handle connections

© 2012 Marty Hall

Questions?

JSF 2, PrimeFaces, Java 7, Ajax, jQuery, Hadoop, RESTful Web Services, Android, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training.

Customized Java EE Training: http://courses.coreservlets.com/
Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.