Network Programming: Servers

Topics in This Section

- **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**

Basics

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);
}
```

   - It is 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**
   - You usually read inside a loop
   - You usually respond in a separate thread
   - Most common way to read input: lines or readLine
   - Most common way to send output: printf

6. **Close the socket when done**

```
server.close();    // Or use try-with-resources
```

   - This closes the associated input and output streams

Reminder of Helper Class: SocketUtils

- **Idea**
  - It is common to make BufferedReader and PrintWriter from a Socket, so simplify the syntax slightly
- **Without SocketUtils (for Socket s)**

  -
    ```
    PrintWriter out =
      new PrintWriter(s.getOutputStream(), true);
    ```

  -
    ```
    BufferedReader in =
      new BufferedReader
        (new InputStreamReader(s.getInputStream()));
    ```
- **With SocketUtils (for Socket s)**
  - `PrintWriter out = SocketUtils.getWriter(s);`
  - `BufferedReader in = SocketUtils.getReader(s);`

Exceptions

- **IOException**
  - Interruption or other unexpected problem
    - Client closing connection causes error for writing, but does *not* cause an error when reading: the Stream<String> from lines just finishes, and null is returned from readLine
- **Note**
  - ServerSocket implements AutoCloseable, so you can use the try-with-resources idea we first covered in file IO section
    - try(ServerSocket listener = new ServerSocket(…)) { … }

Simple Warmup: A Single-Threaded Server

Base Class for 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 or 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);
  }

  @Override
  protected void handleConnection(Socket socket) throws IOException{
    PrintWriter out = SocketUtils.getWriter(socket);
    BufferedReader in = SocketUtils.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 = 8080;
    try {
      port = Integer.parseInt(args[0]);
    } catch(NumberFormatException|ArrayIndexOutOfBoundsException e) {}
    NetworkServerTest tester = new NetworkServerTest(port);
    tester.listen();
  }
}
```

Network Server: Results

- **Accepting a Connection from a browser**
  - Suppose the above test program is started up on port 80 of server.com:

```
> java coreservlets.NetworkServerTest 80
```

  - Then, a standard Web browser on client.com
    requests http://server.com/foo/bar, resulting in the following printed at server.com:

```
Generic Network Server:
got connection from client.com
with first line 'GET /foo/bar HTTP/1.1'
```

A Base Class for a Multithreaded Server

Base Class for Multithreaded 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();
  }
}
```

Inner class whose run method calls back to handleConnection of this class.

The upcoming EchoServer will extend this class to make an HTTP server.

MultithreadedServer.java (Continued – Inner Class)

```
private class ConnectionHandler implements Runnable {
  private Socket connection;

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

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

MultithreadedServer.java (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 connection)
    throws IOException;
```

A Simple Multithreaded HTTP Server

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 for HTTP/1.1)
- If you send HEAD instead of GET, the server returns the same HTTP headers, but no document

Response

```
HTTP/1.1 200 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 extends MultithreadedServer {
  public EchoServer(int port) {
    super(port);
  }

  public static void main(String[] args) {
    int port = 8080;
    try {
      port = Integer.parseInt(args[0]);
    } catch(NumberFormatException|
            ArrayIndexOutOfBoundsException e) {}
    EchoServer server = new EchoServer(port);
    server.listen();
  }
```

EchoServer.java: Reading the Request

```
@Override
public void handleConnection(Socket socket) throws IOException{
  String serverName = "Multithreaded EchoServer";
  PrintWriter out = SocketUtils.getWriter(socket);
  BufferedReader in = SocketUtils.getReader(socket);
  List<String> inputLines = new ArrayList<>();
  String line;
  while((line = in.readLine()) != null) {
    inputLines.add(line);
    if (line.isEmpty()) { // Blank line.
      if (WebUtils.isUsingPost(inputLines)) {
        inputLines.add(WebUtils.postData(in));
      }
      break;
    }
  }
```

EchoServer.java: Sending the Response

```
    WebUtils.printHeader(out, serverName);
    for (String inputLine: inputLines) {
      out.println(inputLine);
    }
    WebUtils.printTrailer(out);
    socket.close();
  }
```

WebUtils.java

```
public static void printHeader(PrintWriter out, String serverName) {
  out.println
    ("HTTP/1.1 200 OK\r\n" +
     "Server: " + serverName + "\r\n" +
     "Content-Type: text/html\r\n" +
     "\r\n" +
     "<!DOCTYPE html>\n" +
     "<html lang=\"en\">\n" +
     "<head>\n" +
     "  <meta charset=\"utf-8\"/>\n" +
     "  <title>" + serverName + " Results</title>\n" +
     "</head>\n" +
     "\n" +
     "<body bgcolor=\"#fdf5e6\">\n" +
     "<h1 align=\"center\">" + serverName + " Results</h1>\n" +
     "Here are the request line and request headers\n" +
     "sent by your browser:\n" +
     "<pre>");
}
```

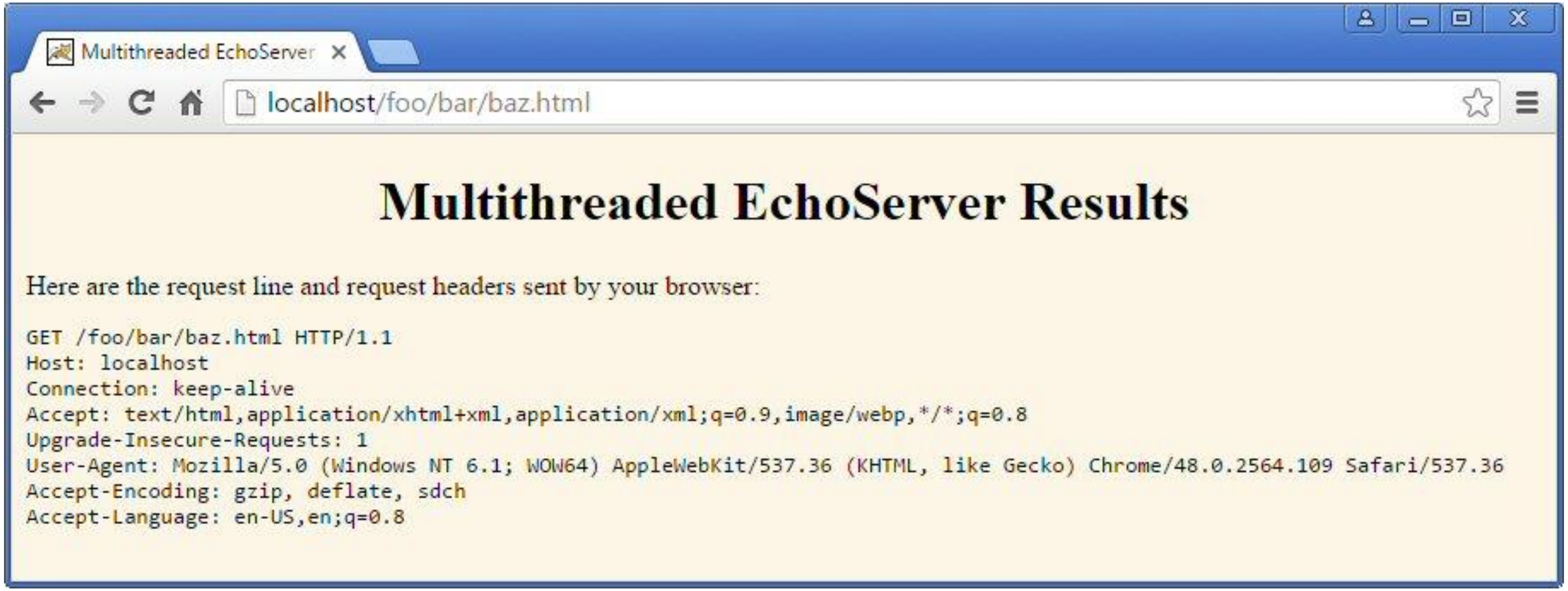
WebUtils.java (Continued)

```
  public static void printTrailer(PrintWriter out) {
    out.println
      ("</pre></body></html>\n");
  }

  public static boolean isUsingPost(List<String> inputs) {
    return(inputs.get(0).toUpperCase().startsWith("POST"));
  }

 /** POST submissions have one extra line at the end, after the blank line,
   *  and NOT terminated by CR. Ignore multi-line posts, such as file uploads. */
  public static String postData(BufferedReader in) throws IOException {
    char[] data = new char[1000];  // Assume 1000 chars max
    int chars = in.read(data);
    return(new String(data, 0, chars));
  }
```

EchoServer in Action



Multithreaded EchoServer Results

Here are the request line and request headers sent by your browser:

```
GET /foo/bar/baz.html HTTP/1.1
Host: localhost
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/48.0.2564.109 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

Wrap-Up

Summary

- **Create a ServerSocket; specify port number**
  - Call accept to wait for a client connection
  - accept returns a Socket object (same class we saw 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/1.1 200 OK),
  - Content-Type (and, optionally, other response headers)
  - Blank line
  - Document
- **Always make servers multi-threaded**
  - Use MultithreadedServer as starting point