Network Programming: Clients

Topics in This Section

- **Overview**
- **Using PrintWriter and BufferedReader**
- **Basic steps for connecting a client to a server**
- **Implementing a generic network client**
- **String formatting and parsing**
  - printf
  - StringTokenizer
  - String.split and regular expressions
- **Problems with blocking I/O**
- **Getting user info from a mail server**
- **Talking to Web servers: general**
- **Talking to Web servers: Java clients**

Overview

Client vs. Server

- **Traditional definition**
  - Client: User of network services
  - Server: Supplier of network services
- **Problem with traditional definition**
  - If there are 2 programs exchanging data, it seems unclear
  - Some situations (e.g., X Windows) seem reversed
- **Easier way to remember distinction**
  - Server starts first. Server doesn't specify host (just port).
  - Client starts second. Client specifies host (and port).
- **Analogy: Company phone line**
  - Installing phone is like starting server
  - Extension is like port
  - Person who calls is the client: he specifies both host (general company number) and port (extension)

Client vs. Server (Continued)

- **If server has to start first, why are we covering clients before we cover servers?**
  - Clients are slightly easier
  - We can test clients by connecting to *existing* servers that are already on the internet
- **Point: clients created in Java need not communicate with servers written in Java.**
  - They can communicate with any server that accepts socket connections (as long as they know the proper communication protocol)
  - Exception: ObjectInputStream and ObjectOutputStream allow Java programs to send complicated data structures back and forth
    - Only works for Java-to-Java communication

Confusing Overuse of the Term "Stream"

- **InputStream**
  - Low-level data structure for reading from socket or file or other input source. We will wrap BufferedReader around it here, and ObjectInputStream around it in later section.
- **OutputStream**
  - Low-level data structure for sending data to socket or file. We will wrap PrintWriter around it here, and ObjectOutputStream around it in later section.
- **Stream<T> (e.g., Stream<String>)**
  - High-level wrapper around arrays, Lists, and other data source. Introduced in Java 8 and covered earlier.
- **IntStream, DoubleStream, etc.**
  - Primitive specializations of Java 8 streams. Covered earlier.

Using a PrintWriter

- **out.printf(formatString, val1, val2, ...);**
  - Takes String-formatting-template and values (one value for each %blah placeholder)
    - See discussion in “Miscellaneous Utilities” section
    - Here, “out” will be connected to a Socket instead of referring to standard output
- **out.print(string) and out.println(string)**
  - Take a single String
- **String.format(formatString, val1, val2, ...);**
  - printf *outputs* a formatted String

```
out.printf("Blah %s%n", 7);
```

  - String.format *returns* a formatted String

```
String s = String.format("Blah %s%n", 7);
out.print(s);
```

Using a BufferedReader

- **in.lines()**
  - Returns a Stream<String>, where each entry in the Stream comes from a separate line in the input. Generally the most powerful and efficient way to read.
    - If socket remains open, you will wait if you try to read to the “end” of the Stream
    - See File I/O section for many examples of usage
- **in.readLine()**
  - Returns a single String
    - If you are sent a String terminated by end-of-line (CR or CR/LF pair), this returns a non-null String
    - If the socket is closed, this returns null
    - If socket is open but no end-of-line, you will wait
- **in.read()**
  - Returns a single character

Basic Steps for a Client to Connect to a Server

Big Idea

- **Connecting to a server is easy**
  - Same basic steps every time
- **But talking to a server can be hard**
  - Depending on the protocol that you define, formatting the requests and parsing the responses may be difficult
- **Steps**
  - Create a Socket
  - Attach a PrintWriter to the Socket (for sending data)
  - Get a BufferedReader from the socket (for reading data)
  - Do I/O (this is only part that changes)
  - Close the socket (automatic with try-with-resources)

Steps for Implementing a Client

1. **Create a Socket object**

```
Socket client = new Socket("hostname", portNumber);
```

2. **Create PrintWriter to send data to the Socket**

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

3. **Create BufferedReader to read response from server**

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

Steps for Implementing a Client (Continued)

4. Do I/O

- PrintWriter
  - printf or println. Summarized in earlier slide.
- BufferedReader
  - lines or readLine. Summarized in earlier slide.

• Notes

- There is standard way to *connect* to server. There is no standard way to *carry on conversation* with server. So, "networking" problems in Java mostly boil down to String formatting problems (for sending data) and String parsing problems (for reading data).
- For Java-to-Java communication, things are much simpler because you can use ObjectInputStream and ObjectOutputStream. See later section on Serialization.

Steps for Implementing a Client (Continued)

5. Close the socket when done

```
client.close();
```

- If you declare the socket using a try-with-resources, this happens automatically, so you do not have to call client.close explicitly
- Closing the socket closes the input and output streams as well

Exceptions

- **UnknownHostException**
  - If host (passed to constructor) not known to DNS server
    - Note that you may use an IP address string for the host
- **IOException**
  - Timeout
  - Connection refused by server
  - Interruption or other unexpected problem
    - Server closing connection does *not* cause read error: null is returned from "readLine", and "lines" simply finishes
- **Note**
  - Socket implements AutoCloseable, so you can use the try-with-resources idea we first covered in file I/O section

```
try(Socket client = new Socket(...)) { ... }
```

Helper Class: SocketUtils

- **Idea**
  - It is common to make BufferedReader and PrintWriter from a Socket, so make minor utility to 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);`
    ```

Helper Class: SocketUtils

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

public class SocketUtils {

  public static BufferedReader getReader(Socket s) throws IOException {
    return(new BufferedReader(
        new InputStreamReader(s.getInputStream())));
  }

  public static PrintWriter getWriter(Socket s) throws IOException {
    // Second argument of true means autoflush
    return(new PrintWriter(s.getOutputStream(), true));
  }

  private SocketUtils() {} // Uninstantiatable class
}
```

A Reusable Network Client Base Class

Idea

- **Connecting to server is almost always the same**
  - Call Socket constructor, use try/catch blocks for UnknownHostException and IOException, close Socket
- **Talking to server is almost always different**
  - Do I/O with PrintWriter and BufferedReader in protocol-specific manner
- **Make reusable NetworkClient class**
  - Automatically does steps that do not change
  - Lets you put in code for the actual I/O (that *does* change)
- **Steps**
  - Extend NetworkClient
  - Override handleConnection(Socket s)
  - Instantiate your class and call connect

A Generic Network Client

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

public abstract class NetworkClient {
  private String host;
  private int port;

  public String getHost() {
    return(host);
  }

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

  /** Register host and port. Connection is established when you call connect. */

  public NetworkClient(String host, int port) {
    this.host = host;
    this.port = port;
  }
```

A Generic Network Client (Continued)

```
  public void connect() {
    try(Socket client = new Socket(host, port)) {
      handleConnection(client);
    } catch(UnknownHostException uhe) {
      System.err.println("Unknown host: " + host);
    } catch(IOException ioe) {
      System.err.println("IOException: " + ioe);
    }
  }

  /** This is the method you will override when
   *  making a network client for your task.
   */
  protected abstract void handleConnection(Socket client) throws IOException;
}
```

Example Client

```
public class NetworkClientTest extends NetworkClient {
  public NetworkClientTest(String host, int port) {
    super(host, port);
  }

  @Override
  protected void handleConnection(Socket client) throws IOException {
    PrintWriter out = SocketUtils.getWriter(client);
    BufferedReader in = SocketUtils.getReader(client);
    out.println("Generic Network Client");
    System.out.printf ("Generic Network Client:%n" +
                       "Connected to '%s' and got '%s' in response.%n",
                        getHost(), in.readLine());
  }
```

Example Client (Continued)

```
  public static void main(String[] args) {
    String host = "localhost";
    int port = 8088;
    if (args.length > 0) {
      host = args[0];
    }
    if (args.length > 1) {
      port = Integer.parseInt(args[1]);
    }
    NetworkClientTest tester = new NetworkClientTest(host, port);
    tester.connect();
  }
}
```

Example Client: Result

```
> java coreservlets.NetworkClientTest ftp.microsoft.com 21
Generic Network Client:
Connected to ftp.microsoft.com and got
'220 Microsoft FTP Service' in response
```

- First line of welcome message from Microsoft's FTP server

```
> java coreservlets.NetworkClientTest djxmmx.net 17
Generic Network Client:
Connected to 'djxmmx.net' and got
'"We have no more right to consume happiness without producing it than
  to' in response.
```

- First line of quote from the "quote of the day" (qotd) server at djxmmx.net
  Find more qotd servers at https://en.wikipedia.org/wiki/QOTD

Aside: String Formatting and Parsing

Formatting and Parsing Strategies

- **Idea**
  - Simple to connect to a server and create Reader/Writer
  - So, hard parts are formatting request and parsing response
- **Approach**
  - Formatting requests
    - Use printf (and String.format) – covered earlier
  - Parsing the response: simple but weak
    - Use StringTokenizer – covered here
  - Parsing the response: more powerful and moderately complicated
    - Use String.split with regular expressions – covered here
  - Parsing the response: most powerful and very complicated
    - Use Pattern and full regex library – not covered

Parsing Strings Using StringTokenizer

- **Idea**
  - Build a tokenizer from an initial string
  - Retrieve tokens one at a time with nextToken
    - A token means a part of the string without the delimeters
  - You can also see how many tokens are remaining (countTokens) or simply test if the number of tokens remaining is nonzero (hasMoreTokens)

```
StringTokenizer tok = new StringTokenizer(input, delimiters);
while (tok.hasMoreTokens()) {
  doSomethingWith(tok.nextToken());
}
```

StringTokenizer

- **Constructors**
  - `StringTokenizer(String input, String delimiters)`
  - `StringTokenizer(String input, String delimiters, boolean includeDelimiters)`
  - `StringTokenizer(String input)`
    - Default delimiter set is " \t\n\r\f" (whitespace)
- **Methods**
  - `nextToken()`
  - `nextToken(String delimiters)`
  - `countTokens()`
  - `hasMoreTokens()`

Interactive Tokenizer: Example

```
import java.util.StringTokenizer;

public class TokTest {
  public static void main(String[] args) {
    if (args.length == 2) {
      String input = args[0], delimiters = args[1];
      StringTokenizer tok = new StringTokenizer(input, delimiters);
      while (tok.hasMoreTokens()) {
        System.out.println(tok.nextToken());
      }
    } else {
      System.out.println
        ("Usage: java TokTest string delimiters");
    }
  }
}
```

Interactive Tokenizer: Result

```
> java coreservlets.TokTest http://www.microsoft.com/~gates/ :/.
http
www
microsoft
com
~gates


> java coreservlets.TokTest "if (tok.hasMoreTokens()) {" "(){. "
if
tok
hasMoreTokens
```

Parsing Strings Using the split Method of String

- **Basic usage**

```
String[] tokens = mainString.split(regexString);
```

- **Differences from StringTokenizer**
  - Entire string is the delimiter (not set of 1-char delimiters as with StringTokenizer)
    - "foobar".split("ob") returns {"fo", "ar"}
    - "foobar".split("bo") returns {"foobar"}
  - You can use regular expressions in the delimiter
    - ^, $, *, +, ., etc for beginning of String, end of String, 0 or more, 1 or more, any one character, etc.
    - See http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#sum
  - Unless you use "+", an empty string is returned between consecutive delimiters
    - "foobar".split("o") returns {"f", "", "bar"}
    - "foobar".split("o+") returns {"f", "bar"}

Importance of Regular Expressions

- **Idea**
  - For Strings, the split, matches, and replaceAll methods use regular expressions.
  - Many other language use regular expressions with similar syntax. Knowing regex syntax is important skill for *every* programmer.

From Randall Munroe and xkcd.com

- **Tutorials**
  - http://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html#sum
  - http://docs.oracle.com/javase/tutorial/essential/regex/
  - http://regexr.com/ (Not Java-specific, but very good)

Interactive String Splitter: Example

```
public class SplitTest {
  public static void main(String[] args) {
    if (args.length == 2) {
      String[] tokens = args[0].split(args[1]);
      for(String token: tokens) {
        if (token.length() != 0) {
          System.out.println(token);
        }
      }
    } else {
      System.out.println("Usage: java SplitTest string delimeters");
    }
  }
}
```

Interactive String Splitter: Result

```
> java coreservlets.TokTest http://www.microsoft.com/~gates/ :/.
http
www
microsoft
com
~gates

> java coreservlets.SplitTest http://www.microsoft.com/~gates/ :/.
http
www.microsoft.com/~gates/


> java coreservlets.SplitTest http://www.microsoft.com/~gates/ [:/.]+
http
www
microsoft
com
~gates
```

This first one is the previous example (using StringTokenizer), for comparison.

Problems with Blocking IO

Aside: Talking to Servers Interactively

- **Telnet**
  - Most people think of telnet as a tool for logging into a remote server on default login port (23). But, it is really more general: a tool for connecting to a remote server on any port and interactively sending commands and looking at results
  - Before you write a Java client to connect to a server, telnet to the server, try out commands interactively, and see if the protocol works the way you think it does
- **Enabling telnet on Windows 7, 8, or 10**
  - Starting with Windows Vista, telnet is disabled by default
    - Google "install telnet windows" to see how to enable it
    - You may also need to turn on local echo
    - Unix telnet clients are *much* more convenient
    - Putty is a popular free telnet client for Windows

Using Telnet to Verify Email Addresses

- **Example talking to SMTP server**

```
> telnet apl.jhu.edu 25
Trying 128.220.101.100 ...Connected …
220 aplcenmp.apl.jhu.edu Sendmail …
expn hall
250 Marty Hall <hall@aplcenmp.apl.jhu.edu>
expn root
250 Gary Gafke <…>
250 Tom Vellani <…>
quit
221 aplcenmp.apl.jhu.edu closing connection
Connection closed by foreign host.
```

A Client to Verify Email Addresses

```
public class AddressVerifier extends NetworkClient {
  private String username;

  public AddressVerifier(String username, String hostname,
                         int port) {
    super(hostname, port);
    this.username = username;
  }

  public static void main(String[] args) {
    if (args.length != 1) {
      usage();
    }
    MailAddress address = new MailAddress(args[0]);
    new AddressVerifier(address.getUsername(),
                        address.getHostname(), 25);
  }
```

A Client to Verify Email Addresses (Continued)

```
protected void handleConnection(Socket client)
    throws IOException {
  PrintWriter out = SocketUtils.getWriter(client);
  InputStream rawIn = client.getInputStream();
  byte[] response = new byte[1000];
  // Clear out mail server's welcome message.
  rawIn.read(response);
  out.println("EXPN " + username);
  // Read the response to the EXPN command.
  int numBytes = rawIn.read(response);
  // The 0 means to use normal ASCII encoding.
  System.out.write(response, 0, numBytes);
  out.println("QUIT");
}
...
}
```

Main point: you can only use readLine if either

- You know how many lines of data will be sent (call readLine that many times)
- The server will close the connection when done, as with HTTP servers (call readLine until you get null)

Helper Class: MailAddress

```
public class MailAddress {
  private String username, hostname;

  public MailAddress(String emailAddress) {
    String[] pieces = emailAddress.split("@");
    if (pieces.length != 2) {
      System.out.println("Illegal email address");
      System.exit(-1);
    } else {
      username = pieces[0];
      hostname = pieces[1];
    }
  }

  public String getUsername() { return(username); }

  public String getHostname() { return(hostname); }
}
```

Address Verifier: Result

```
> java AddressVerifier tbl@w3.org
250 <timbl@hq.lcs.mit.edu>

> java AddressVerifier timbl@hq.lcs.mit.edu
250 Tim Berners-Lee <timbl>

> java AddressVerifier gosling@mail.javasoft.com
550 gosling... User unknown
```

Talking to Web Servers: General

Brief Aside: The HTTP "GET" Command

For URL http://somehost/somepath

HTTP 1.0	HTTP 1.1

Connect to somehost on port 80
`GET /somepath HTTP/1.0`
Blank line
	Connect to somehost on port 80
	`GET /somepath HTTP/1.1`
	`Host: somehost`
	`Connection: close`
	Blank line

Notes:

- The server closes the connection when it is done sending the page. This makes it easy for Java to read the page: it can just use lines() and then process the entire Stream<String>, or use readLine() until it gets null.
- The reason for the Host header in HTTP 1.1 is so that you can connect to hosts that are using virtual hosting, i.e., sites that host multiple domain names on the same machine. Most small and medium sites use virtual hosting.
- The address after "GET" is called a URI.
- Regarding top-level URLs: for the URL http://somehost, the browser treats it as though it were http://somehost/, and sends / as the URI for the GET request.

Example: Telnetting to Web Server

URL: http://www.apl.jhu.edu/~hall/

```
Unix> telnet www.apl.jhu.edu 80
Trying 128.220.101.100 ...
Connected to aplcenmp.apl.jhu.edu.
Escape character is '^]'.
GET /~hall/ HTTP/1.0

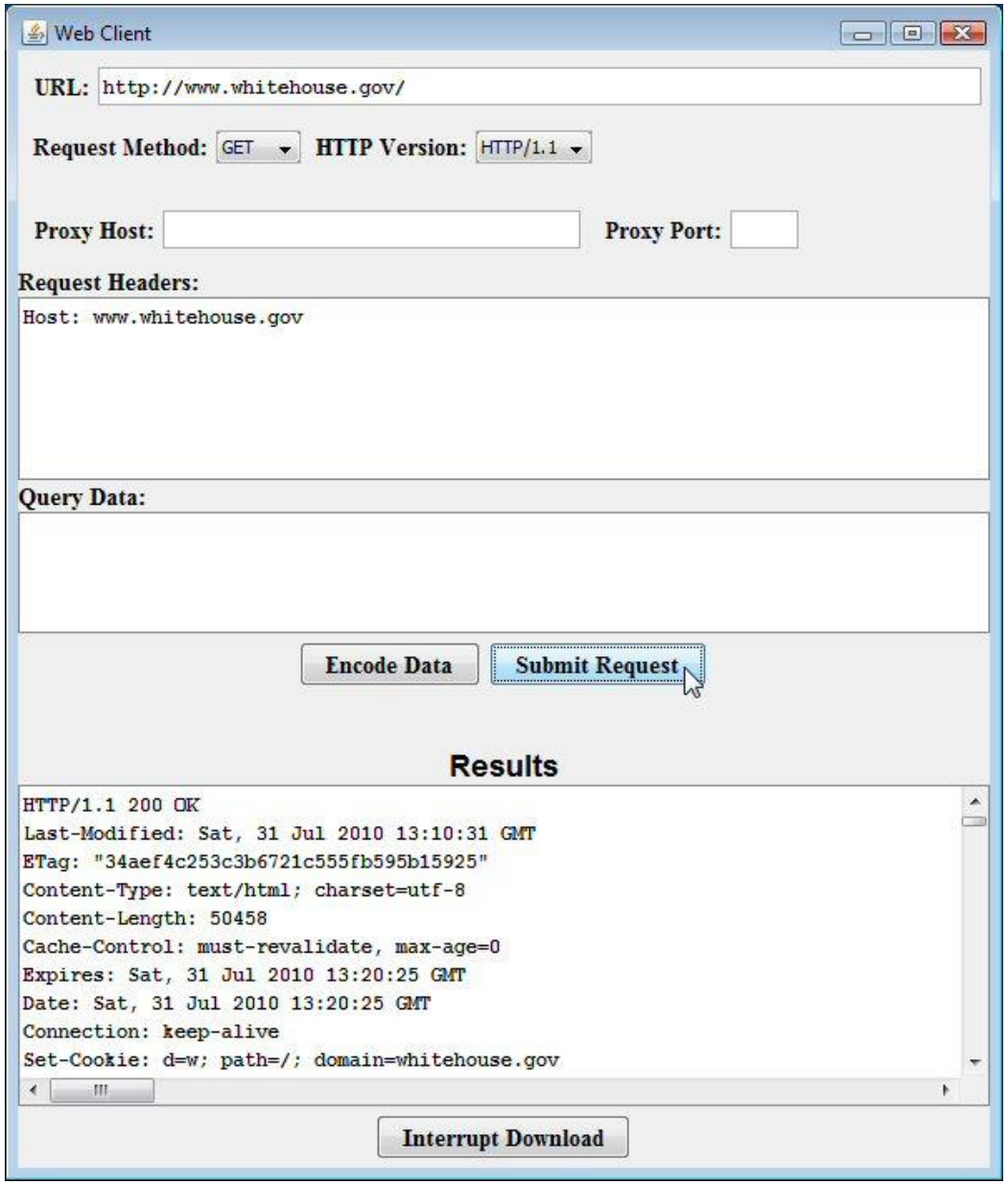
HTTP/1.1 200 OK
...
Connection: close
Content-Type: text/html; charset=ISO-8859-1

<!DOCTYPE HTML>
<html>
...
</html>Connection closed by foreign host.
Unix>
```

Talking to Web Servers Interactively

- **Problem**
  - MS Windows telnet client works poorly for this
    - Linux, Solaris, and MacOS telnet clients work fine for this
- **Solution: WebClient**
  - Simple graphical user interface to communicate with HTTP servers
  - User can interactively specify:
    - URL with host, port, and URI
    - HTTP request headers
  - HTTP request is performed in a separate thread
  - Response document is placed in a scrollable text area
  - Download all source files for WebClient from tutorial home page

WebClient: Example



Talking to Web Servers: A Java Client

Retrieving a URI from a Host

```
public class UriRetriever extends NetworkClient {
  private String uri;

  public static void main(String[] args) {
    UriRetriever retriever =
      new UriRetriever(args[0], Integer.parseInt(args[1]), args[2]);
    retriever.connect();
  }

  public UriRetriever(String host, int port, String uri) {
    super(host, port);
    this.uri = uri;
  }
```

Retrieving a URI from a Host

```
  // It is safe to use in.lines() or loop and
  // do in.readLine() because Web servers close
  // the connection after sending the data.

  protected void handleConnection(Socket client) throws IOException {
    PrintWriter out = SocketUtils.getWriter(client);
    BufferedReader in = SocketUtils.getReader(client);
    out.printf("GET %s HTTP/1.1\r\n", uri);
    out.printf("Host: %s\r\n", getHost());
    out.printf("Connection: close\r\n\r\n");
    in.lines().forEach(System.out::println);
  }
}
```

Echoing All Lines: Java 8 vs. Java 7

- **Java 8 (previous slide)**

```
in.lines().forEach(System.out::println);
```

- **Java 7**

```
String line;
while ((line = in.readLine()) != null) {
  System.out.println(line);
}
```

Helper Class: Parsing a URL

```
public class UrlParser {
  private String host;
  private int port = 80;
  private String uri;

  public UrlParser(String url) {
    StringTokenizer tok = new StringTokenizer(url);
    String protocol = tok.nextToken(":");
    checkProtocol(protocol);
    host = tok.nextToken(":/");
    try {
      uri = tok.nextToken("");
      if (uri.charAt(0) == ':') {
        tok = new StringTokenizer(uri);
        port = Integer.parseInt(tok.nextToken(":/"));
        uri = tok.nextToken("");
      }
    } catch(NoSuchElementException nsee) {
      uri = "/";
    } ...
```

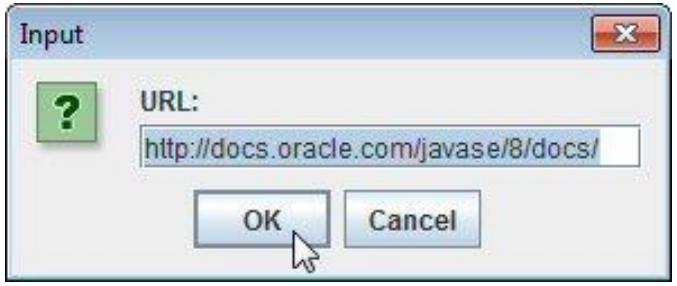
A Class to Retrieve a Given URL

```
public class UrlRetriever {
  public static void main(String[] args) {
    String defaultAddress = "http://docs.oracle.com/javase/8/docs/";
    String address = JOptionPane.showInputDialog("URL:", defaultAddress);
    if (address == null) {
      address = defaultAddress;
    }
    UrlParser parser = new UrlParser(address);
    UriRetriever uriClient =
      new UriRetriever(parser.getHost(), parser.getPort(), parser.getUri());
    uriClient.connect();
  }
}
```

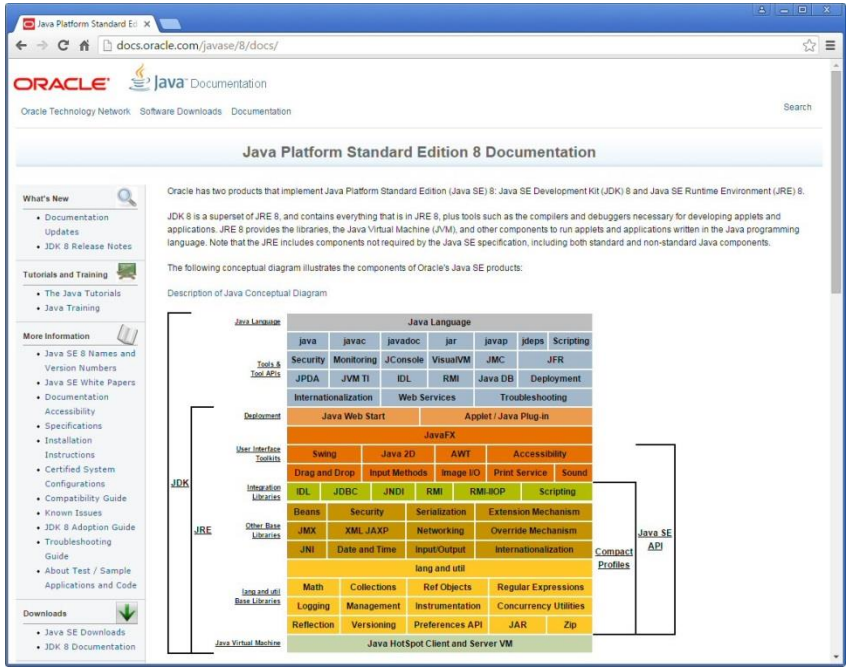
UrlRetriever in Action

- **No explicit port number**

```
HTTP/1.1 200 OK
Server: Apache
...
```




```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE HTML …>
<html ...>
<head>...</head>
<body>
...
</body>
</html>
```

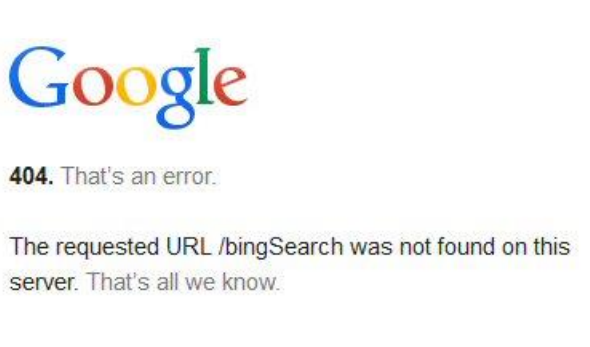


UrlRetriever in Action (Continued)

- **Explicit port number**

```
HTTP/1.1 404 Not Found
Content-Type: text/html
...

<!DOCTYPE html>
<html lang=en>
...
<p><b>404.</b> <ins>That's an error.</ins>
<p>The requested URL <code>/bingSearch</code>
was not found on this server.
<ins>That's all we know.</ins>
...
```



Talking to Web Servers: Using the URL Class

Writing a Web Browser

- **Wow! We just wrote a browser in 3 pages**
  - Well, not exactly, since we just show the raw HTML, not format it or handle interactions with the end user
  - Still, not bad for such concise code

- **We can do better**
  - It is important to understand how to connect to servers explicitly, since in real life you will probably be writing custom servers with custom protocols
  - But, for the specific case of connecting to a Web server, Java has builtin support
    - Make a URL
    - Call openConnection
    - Wrap a BufferedReader around an InputStreamReader, around that result
    - Use lines or readLine

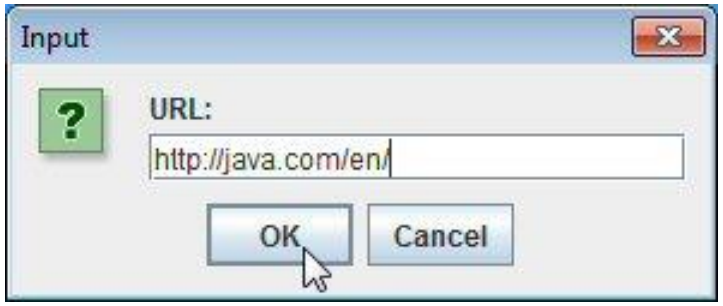
Helper Method: Getting Lines of a Page

```
public class WebUtils {
  public static Stream<String> lines(String address) {
    try {
      URL url = new URL(address);
      BufferedReader in =
        new BufferedReader(new InputStreamReader(url.openStream()));
      return(in.lines());
    } catch(IOException ioe) {
      System.err.println(ioe);
      return(Stream.empty());
    }
  }
  ...
}
```

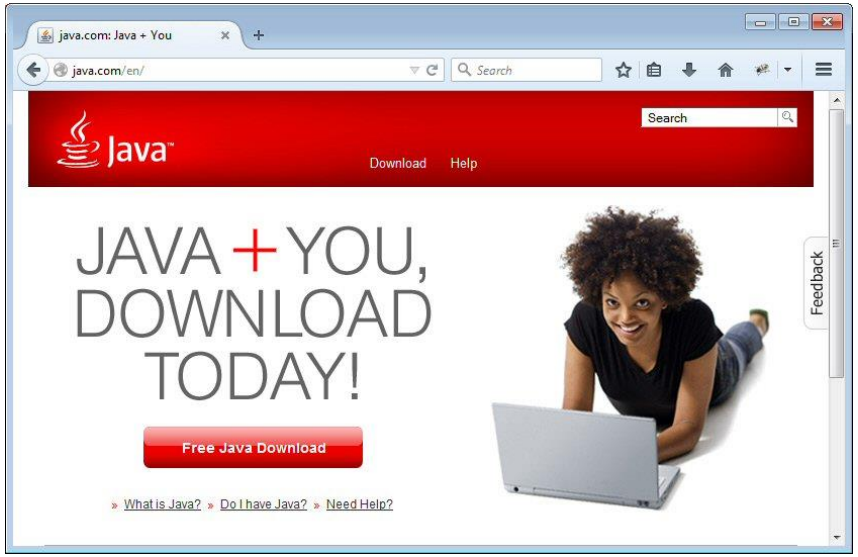
Browser in 1 Page: Using URL

```
public class UrlRetriever2 {
  public static void main(String[] args) {
    String defaultAddress = "http://docs.oracle.com/javase/8/docs/";
    String address =
      JOptionPane.showInputDialog("URL:", defaultAddress);
    if (address == null) {
      address = defaultAddress;
    }
    WebUtils.lines(address).forEach(System.out::println);
  }
}
```

UrlRetriever2 in Action




```
<!DOCTYPE HTML ...>
<html lang="en-US" xml:lang="en-US">
...
<h1>Java<em>+</em>You, Download Today!</h1>
...
</html>
```



Useful URL Methods

- **openConnection**
  - Yields a `URLConnection` that establishes a connection to host specified by the URL
  - Used to retrieve header lines and to supply data to the HTTP server
- **openInputStream**
  - Returns the connection's input stream for reading
- **toExernalForm**
  - Gives the string representation of the URL
- **getRef, getFile, getHost, getProtocol, getPort**
  - Returns the different components of the URL

Using the URL Methods: Example

```
public class UrlTest {
  public static void main(String[] args) {
    if (args.length == 1) {
      try {
        URL url = new URL(args[0]);
        System.out.println
          ("URL: " + url.toExternalForm() + "\n" +
           "  File:      " + url.getFile() + "\n" +
           "  Host:      " + url.getHost() + "\n" +
           "  Port:      " + url.getPort() + "\n" +
           "  Protocol:  " + url.getProtocol() + "\n" +
           "  Reference: " + url.getRef());
      } catch(MalformedURLException mue) {
        System.out.println("Bad URL.");
      }
    } else
      System.out.println("Usage: UrlTest <URL>");
  }
}
```

Using the URL Methods, Result

```
> java coreservlets.UrlTest http://www.irs.gov/mission/#squeezing-them-dry
URL: http://www.irs.gov/mission/#squeezing-them-dry
  File:      /mission/
  Host:      www.irs.gov
  Port:      -1
  Protocol:  http
  Reference: squeezing-them-dry
```

Note: If the port is not explicitly stated in the URL, then the standard port for the protocol is assumed, and getPort returns –1

A Real Browser Using Swing

- The `JEditorPane` class has builtin support for HTTP and HTML

Browser in Swing: Code

```
import javax.swing.*;
import javax.swing.event.*;
...

public class Browser extends JFrame implements HyperlinkListener, ActionListener {
  private JEditorPane htmlPane;
  ...

  public Browser(String initialURL) {
    ...
    try {
        htmlPane = new JEditorPane(initialURL);
        htmlPane.setEditable(false);
        htmlPane.addHyperlinkListener(this);
        JScrollPane scrollPane = new JScrollPane(htmlPane);
        getContentPane().add(scrollPane, BorderLayout.CENTER);
    } catch(IOException ioe) {
       warnUser("Can't build HTML pane for " + initialURL + ": " + ioe);
    }
```

Browser in Swing (Continued)

```
    ...
    Dimension screenSize = getToolkit().getScreenSize();
    int width = screenSize.width * 8 / 10;
    int height = screenSize.height * 8 / 10;
    setBounds(width/8, height/8, width, height);
    setVisible(true);
  }

  public void actionPerformed(ActionEvent event) {
    String url;
    if (event.getSource() == urlField)
      url = urlField.getText();
    else // Clicked "home" button instead of entering URL
      url = initialURL;
    try {
      htmlPane.setPage(new URL(url));
      urlField.setText(url);
    } catch(IOException ioe) {
      warnUser("Can't follow link to " + url + ": " + ioe);
    }
  }
```

Browser in Swing (Continued)

```
...
public void hyperlinkUpdate(HyperlinkEvent event) {
  if (event.getEventType() ==
                HyperlinkEvent.EventType.ACTIVATED) {
    try {
      htmlPane.setPage(event.getURL());
      urlField.setText(event.getURL().toExternalForm());
    } catch(IOException ioe) {
       warnUser("Can't follow link to "
                + event.getURL().toExternalForm() + ": " + ioe);
    }
  }
}
```

Wrap-Up

Summary

- **Open a Socket**

```
new Socket("hostname-or-IP-Address", port)
```

- **Get a PrintWriter to use for sending data to server**

```
new PrintWriter(client.getOutputStream(), true)
```

- **Get a BufferedReader to use for reading data from server**

```
new BufferedReader
      (new InputStreamReader(client.getInputStream()))
```

- **Notes**
  - lines and readLine block until data is received or connection is closed
    - If connection is closed, lines just completes the Stream and readLine returns null
  - HTTP servers normally close the connection after sending data, so it is safe to use lines or readLine
  - String.split and StringTokenizer help parse strings