# COURSE OUTCOMES(CO’S)

a) Develop Programs using GUI Framework (AWT and Swings)

b) Handle events of AWT and Swings Components.

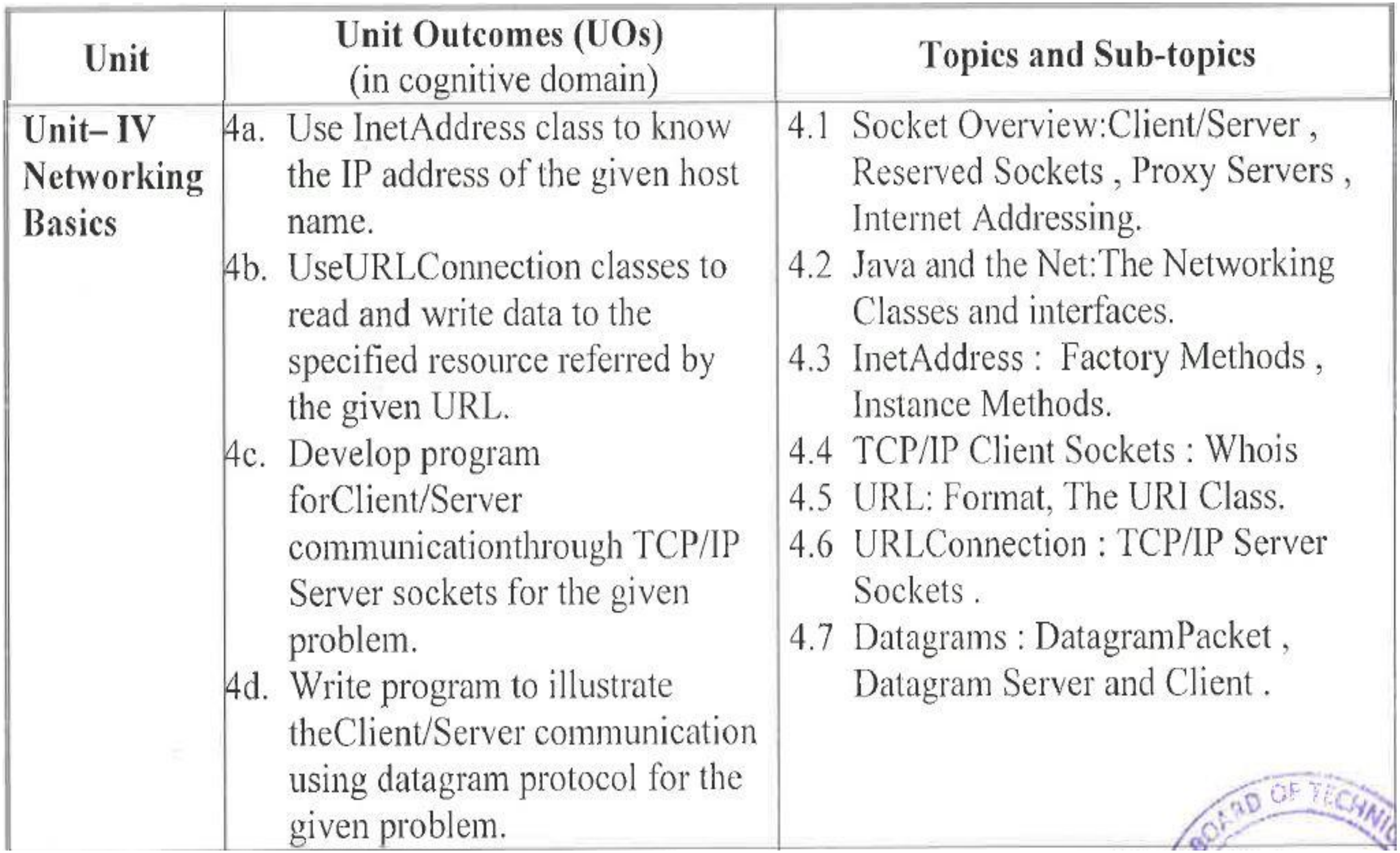
c) Develop programs to handle events in Java Programming

d) Develop Java programs using networking basics.

e) Develop programs using Database.

**f) Develop programs using Servlets.**

# UNIT OUTCOMES(UO'S)



| Unit | Unit Outcomes (UOs)<br>(in cognitive domain) | Topics and Sub-topics |
|---|---|---|
| **Unit– IV Networking Basics** | 4a. Use InetAddress class to know the IP address of the given host name.<br>4b. UseURLConnection classes to read and write data to the specified resource referred by the given URL.<br>4c. Develop program forClient/Server communicationthrough TCP/IP Server sockets for the given problem.<br>4d. Write program to illustrate theClient/Server communication using datagram protocol for the given problem. | 4.1 Socket Overview:Client/Server , Reserved Sockets , Proxy Servers , Internet Addressing.<br>4.2 Java and the Net:The Networking Classes and interfaces.<br>4.3 InetAddress : Factory Methods , Instance Methods.<br>4.4 TCP/IP Client Sockets : Whois<br>4.5 URL: Format, The URI Class.<br>4.6 URLConnection : TCP/IP Server Sockets .<br>4.7 Datagrams : DatagramPacket , Datagram Server and Client . |

# NETWORKING BASICS

- Java is practically a synonym for Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, cross platform, portable code.

- However, one of the most important reasons that Java is the premier language for network programming are the classes defined in the **java.net** package.

- They provide an easy-to-use means by which programmers of all skill levels can access network resources.

- This chapter explores the **java.net** package. It is important to emphasize that networking is a very large and at times complicated topic.

# Networking Basics

- At the core of Java’s networking support is the concept of a *socket*. A socket identifies an endpoint in a network.
- The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term *Berkeley socket* is also used.
- Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information.
- This is accomplished through the use of a *port*, which is a numbered socket on a particular machine.

# Networking Basics

- A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique.

- To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

# NETWORKING BASICS

- **Socket communication takes place via a protocol.**
- ***Internet Protocol (IP)*** is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.
- ***Transmission Control Protocol* (TCP)** is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data.
- A third protocol, ***User Datagram Protocol (UDP),*** sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

# NETWORKING BASICS

- Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols.

- Many of these will seem familiar to you if you have spent any time surfing the Internet.
  1. Port number 21 is for FTP;
  2. 23 is for Telnet;
  3. 25 is for e-mail;
  4. 43 is for whois;
  5. 80 is for HTTP; 1
  6. 19 is for netnews—and the list goes on.

- It is up to each protocol to determine how a client should interact with the port.

# Networking Basics

- **For example**, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images.

- It is a quite simple protocol for a basic page-browsing web server. Here’s how it works.

- **When a client requests a file from an HTTP server, an action known as a *hit*,** it simply sends the name of the file in a special format to a predefined port and reads back the contents of the file.

- The server also responds with a status code to tell the client whether or not the request can be fulfilled and why.

# NETWORKING BASICS

- A key component of the Internet is the ***address*.** Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net.

- Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values.

- This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play.

- **IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks.**

- Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4.

# NETWORKING BASICS

- Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its *domain name*, describes a machine’s location in a name space.

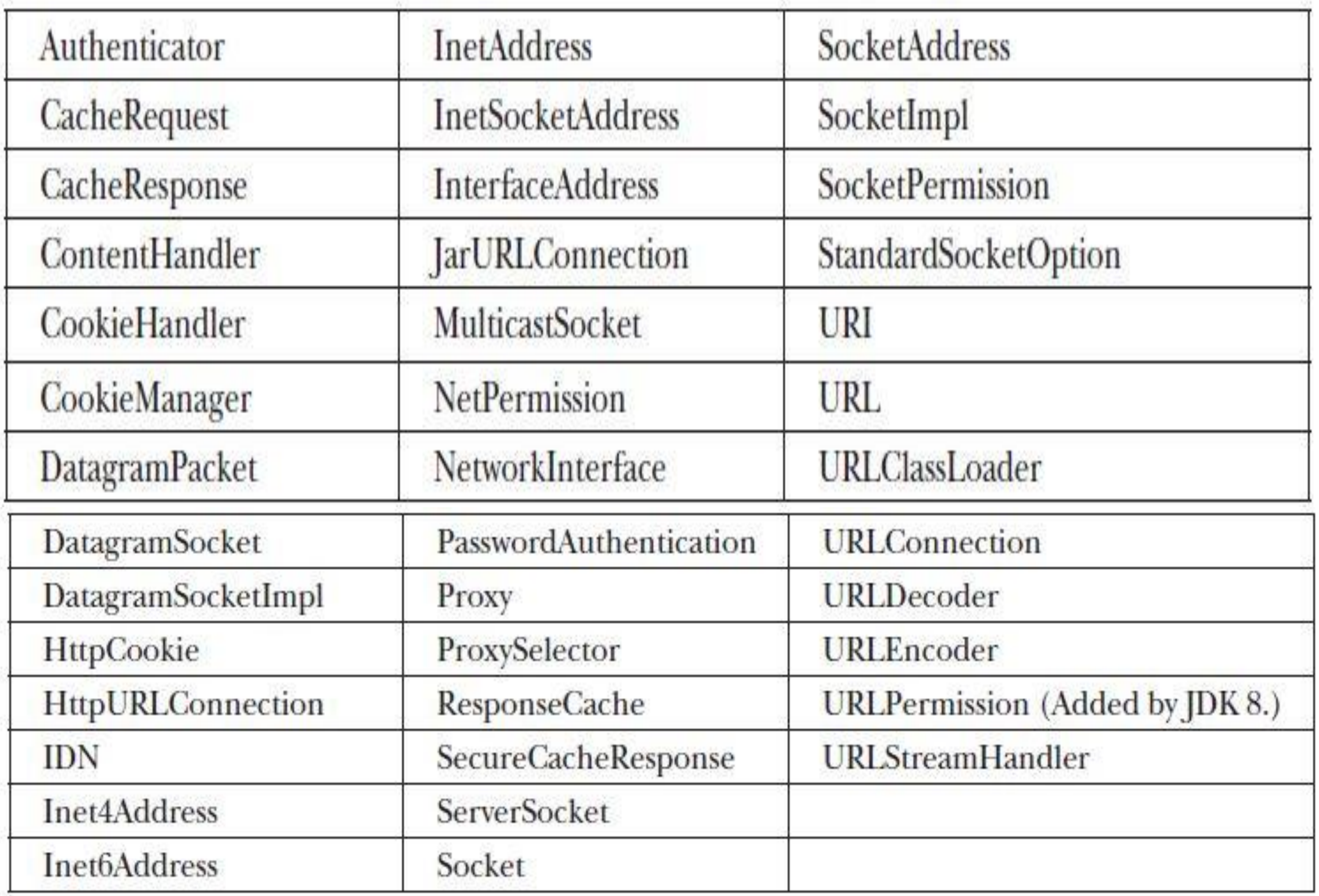
- For example, **www.HerbSchildt.com** is in the *COM* top-level domain (reserved for U.S. commercial sites); it is called *HerbSchildt*, and *www* identifies the server for web requests.

- **An Internet domain name is mapped to an IP address by the *Domain Naming Service (DNS).***

- This enables users to work with domain names, but the Internet operates on IP addresses.

# The Networking Classes and Interfaces

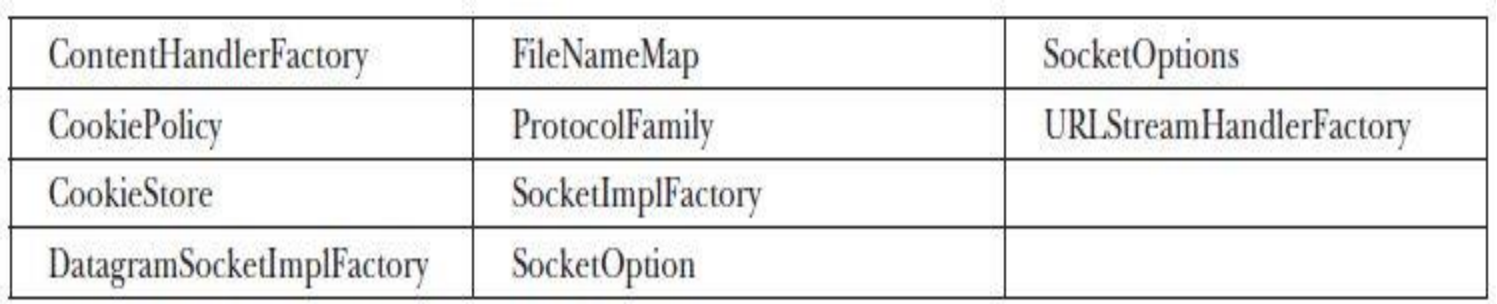
- The classes contained in the **java.net** package are shown here:



| | | |
|---|---|---|
| Authenticator | InetAddress | SocketAddress |
| CacheRequest | InetSocketAddress | SocketImpl |
| CacheResponse | InterfaceAddress | SocketPermission |
| ContentHandler | JarURLConnection | StandardSocketOption |
| CookieHandler | MulticastSocket | URI |
| CookieManager | NetPermission | URL |
| DatagramPacket | NetworkInterface | URLClassLoader |
| DatagramSocket | PasswordAuthentication | URLConnection |
| DatagramSocketImpl | Proxy | URLDecoder |
| HttpCookie | ProxySelector | URLEncoder |
| HttpURLConnection | ResponseCache | URLPermission (Added by JDK 8.) |
| IDN | SecureCacheResponse | URLStreamHandler |
| Inet4Address | ServerSocket | |
| Inet6Address | Socket | |

# The Networking Classes and Interfaces

- The **java.net** package’s interfaces are listed here:



| | | |
|---|---|---|
| ContentHandlerFactory | FileNameMap | SocketOptions |
| CookiePolicy | ProtocolFamily | URLStreamHandlerFactory |
| CookieStore | SocketImplFactory | |
| DatagramSocketImplFactory | SocketOption | |

# InetAddress

- The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address.

- You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address.

- The **InetAddress** class hides the number inside. **InetAddress** can handle both IPv4 and IPv6 addresses.

- **Inet4Address and Inet6Address**

- Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclasses of **InetAddress** were created: **Inet4Address** and **Inet6Address**.

- **Inet4Address** represents a traditional-style IPv4 address. **Inet6Address** encapsulates a newer IPv6 address.

# Factory Methods

- The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you have to use one of the available factory methods.
- ***Factory methods*** are merely a convention whereby static methods in a class return an instance of that class.
- This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer.

# Factory Methods

- Three commonly used **InetAddress** factory methods are shown here:

- **static InetAddress getLocalHost( )**

  throws UnknownHostException

- **static InetAddress getByName(String *hostName*)**

  throws UnknownHostException

- **static InetAddress[ ] getAllByName(String *hostName*)**

  throws UnknownHostException

# Factory Methods

- The **getLocalHost( )** method simply returns the **InetAddress** object that represents the localhost.

- The **getByName( )** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name, they throw an **UnknownHostException**.

# Factory Methods

- On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling.

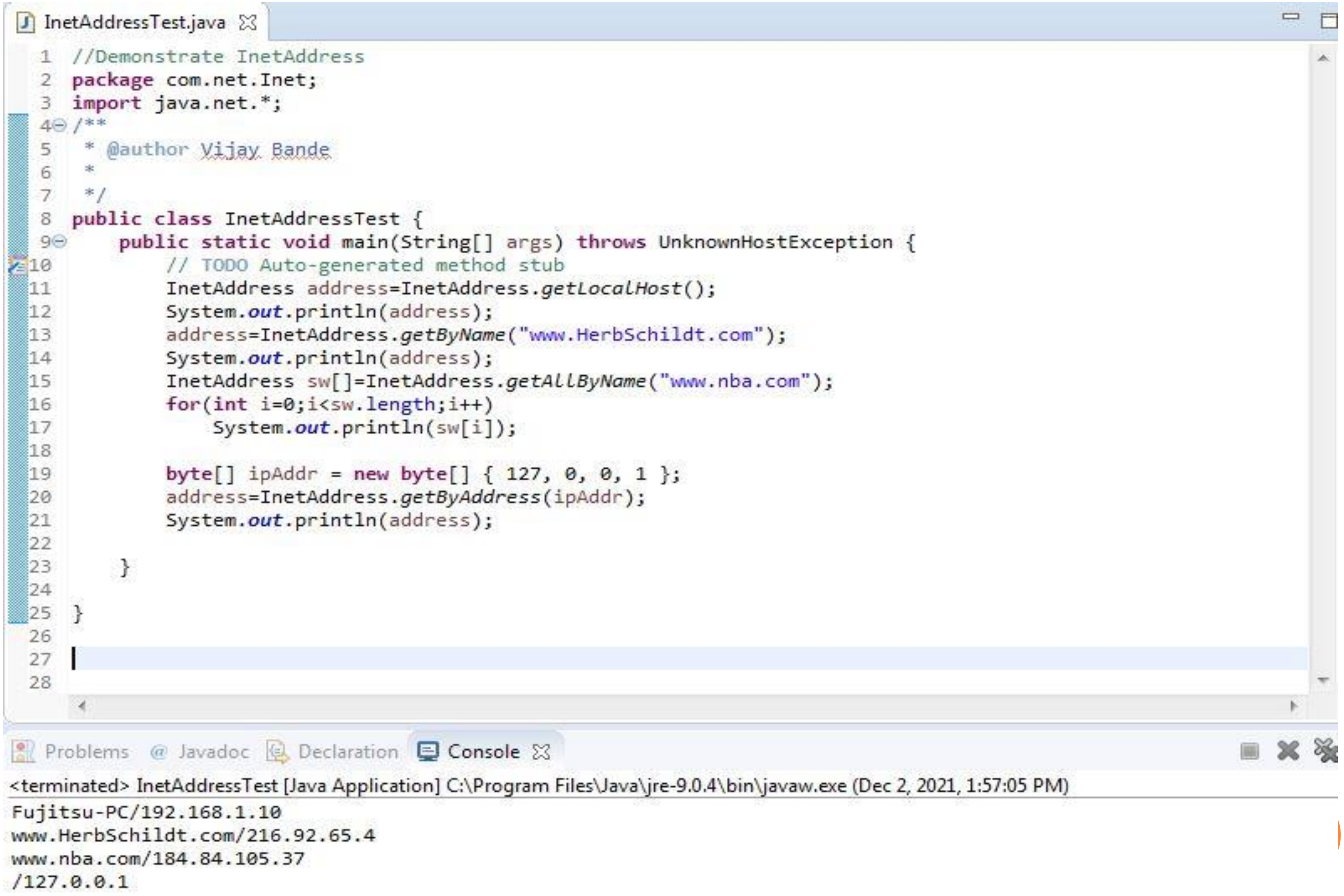
- The **getAllByName( )** factory method returns an array of **InetAddresses** that represent all of the addresses that a particular name resolves to. It will also throw an **UnknownHostException** if it can't resolve the name to at least one address.

- **InetAddress** also includes the factory method **getByAddress( ),** which takes an IP address and returns an **InetAddress** object. Either an IPv4 or an IPv6 address can be used.

- **static** InetAddress getByAddress(**byte**[] addr)

  **throws** UnknownHostException

# INETADDRESS EXAMPLE

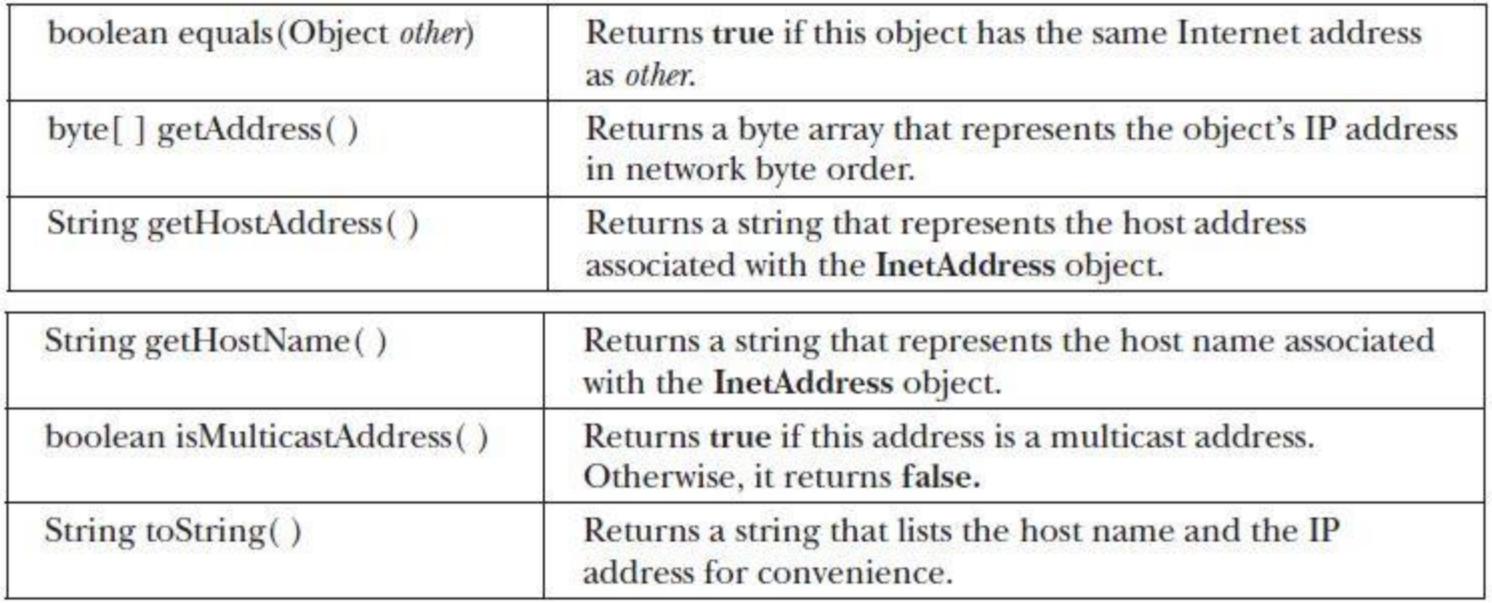


```java
//Demonstrate InetAddress
package com.net.Inet;
import java.net.*;
/**
 * @author Vijay Bande
 *
 */
public class InetAddressTest {
    public static void main(String[] args) throws UnknownHostException {
        // TODO Auto-generated method stub
        InetAddress address=InetAddress.getLocalHost();
        System.out.println(address);
        address=InetAddress.getByName("www.HerbSchildt.com");
        System.out.println(address);
        InetAddress sw[]=InetAddress.getAllByName("www.nba.com");
        for(int i=0;i<sw.length;i++)
            System.out.println(sw[i]);

        byte[] ipAddr = new byte[] { 127, 0, 0, 1 };
        address=InetAddress.getByAddress(ipAddr);
        System.out.println(address);

    }

}

```

```
<terminated> InetAddressTest [Java Application] C:\Program Files\Java\jre-9.0.4\bin\javaw.exe (Dec 2, 2021, 1:57:05 PM)
Fujitsu-PC/192.168.1.10
www.HerbSchildt.com/216.92.65.4
www.nba.com/184.84.105.37
/127.0.0.1
```

# Instance Methods

- Here are some of the more commonly used methods:



| | |
|---|---|
| boolean equals(Object *other*) | Returns **true** if this object has the same Internet address as *other.* |
| byte[ ] getAddress( ) | Returns a byte array that represents the object's IP address in network byte order. |
| String getHostAddress( ) | Returns a string that represents the host address associated with the **InetAddress** object. |
| String getHostName( ) | Returns a string that represents the host name associated with the **InetAddress** object. |
| boolean isMulticastAddress( ) | Returns **true** if this address is a multicast address. Otherwise, it returns **false.** |
| String toString( ) | Returns a string that lists the host name and the IP address for convenience. |

# TCP/IP Client Sockets

- TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet.

- A socket can be used to connect Java’s I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

# TCP/IP Client Sockets

- **There are two kinds of TCP sockets in Java**. One is for servers, and the other is for clients.

- The **ServerSocket** class is designed to be a "listener," which waits for clients to connect before doing anything. Thus**, ServerSocket is for servers**. **The Socket class is for clients.**

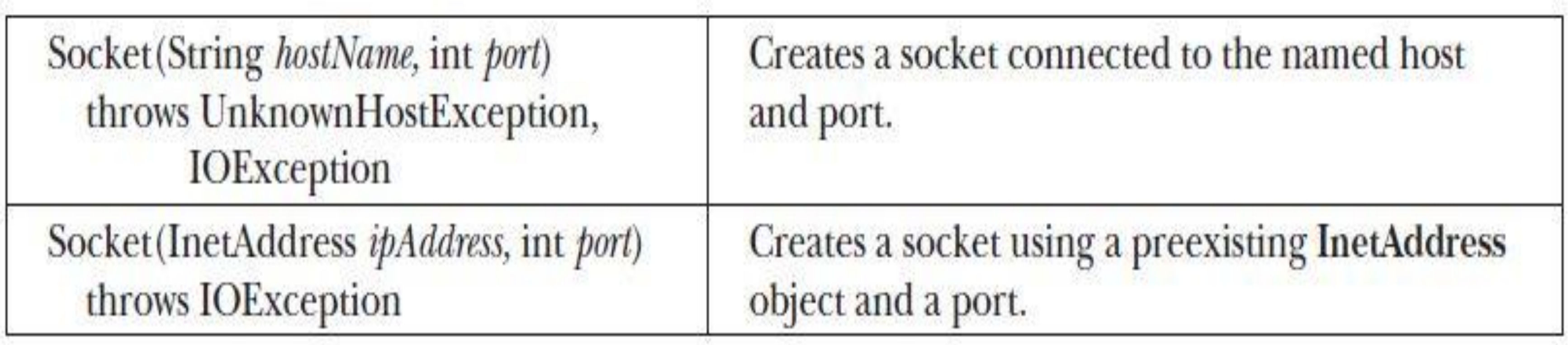
- It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications.

# TCP/IP Client Sockets

- The creation of a **Socket** object implicitly establishes a connection between the client and server.

- There are no methods or constructors that explicitly expose the details of establishing that connection.

- Here are two constructors used to create client sockets:

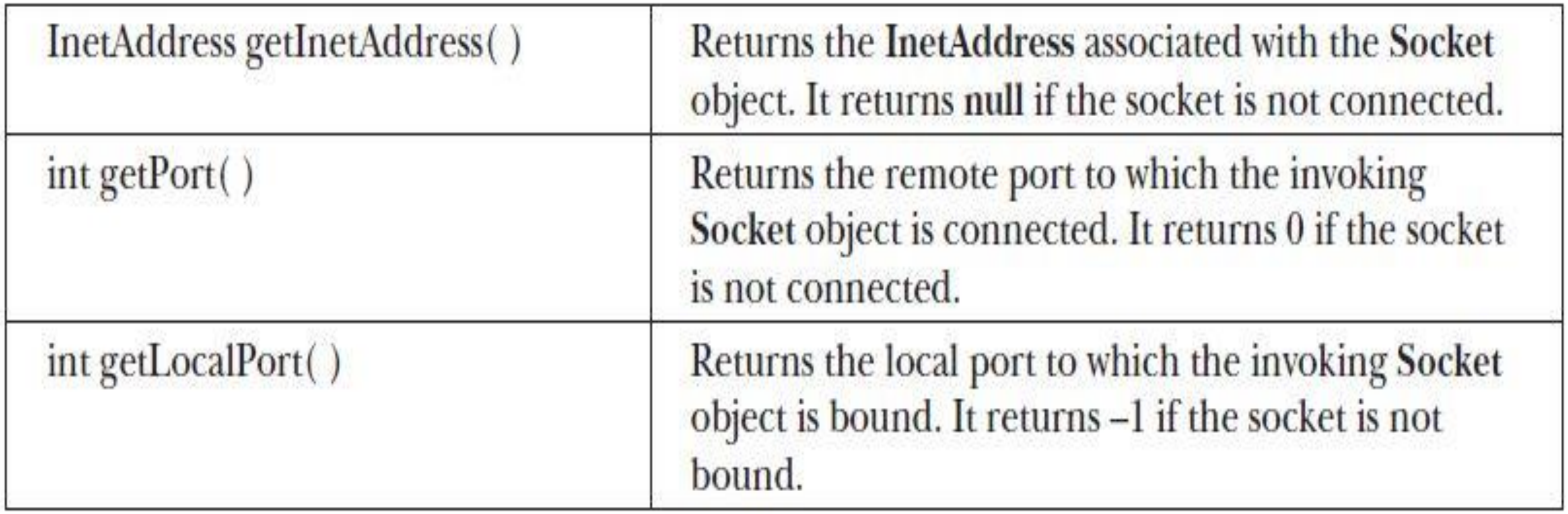


| | |
|---|---|
| Socket(String *hostName*, int *port*)<br>throws UnknownHostException,<br>IOException | Creates a socket connected to the named host and port. |
| Socket(InetAddress *ipAddress*, int *port*)<br>throws IOException | Creates a socket using a preexisting **InetAddress** object and a port. |

# TCP/IP Client Sockets

- **Socket** defines several instance methods.

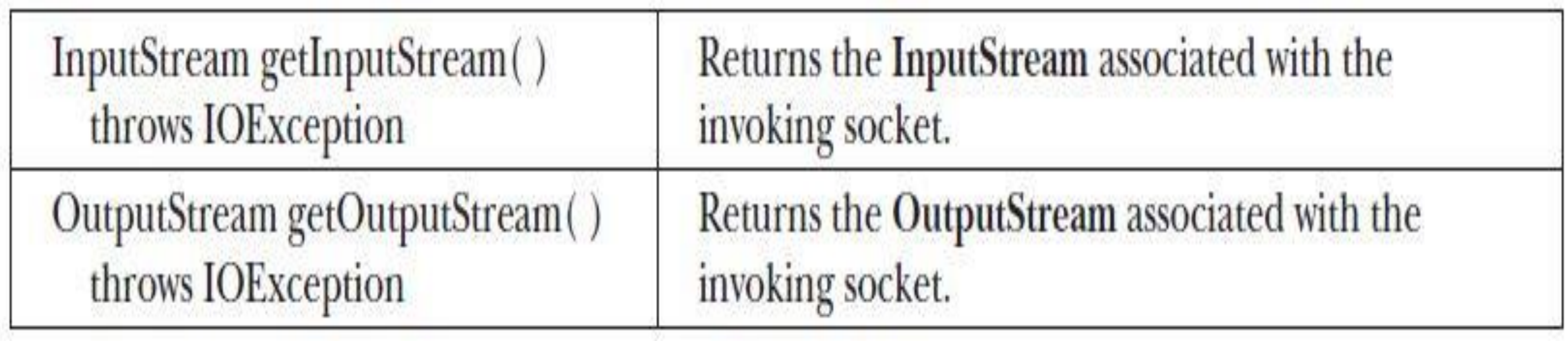
- For example, a **Socket** can be examined at any time for the address and port information associated with it, by use of the following methods:



| | |
|---|---|
| InetAddress getInetAddress( ) | Returns the **InetAddress** associated with the **Socket** object. It returns **null** if the socket is not connected. |
| int getPort( ) | Returns the remote port to which the invoking **Socket** object is connected. It returns 0 if the socket is not connected. |
| int getLocalPort( ) | Returns the local port to which the invoking **Socket** object is bound. It returns –1 if the socket is not bound. |

# TCP/IP Client Sockets

- You can gain access to the input and output streams associated with a **Socket** by use of the **getInputStream( )** and **getOuptutStream( )** methods, as shown here.



| | |
|---|---|
| InputStream getInputStream( ) throws IOException | Returns the **InputStream** associated with the invoking socket. |
| OutputStream getOutputStream( ) throws IOException | Returns the **OutputStream** associated with the invoking socket. |

- Each can throw an **IOException** if the socket has been invalidated by a loss of connection.

# TCP/IP Client Sockets

- Several other methods are available, including
  - **connect( ),** which allows you to specify a new connection;
  - **isConnected( ),** which returns true if the socket is connected to a server;
  - **isBound( ),** which returns true if the socket is bound to an address; and
  - **isClosed( ),** which returns true if the socket is closed.
  - To close a socket, call **close( )**. Closing a socket also closes the I/O streams associated with the socket.
- Beginning with JDK 7, **Socket** also implements **AutoCloseable**, which means that you can use a **try**-with-resources block to manage a socket.

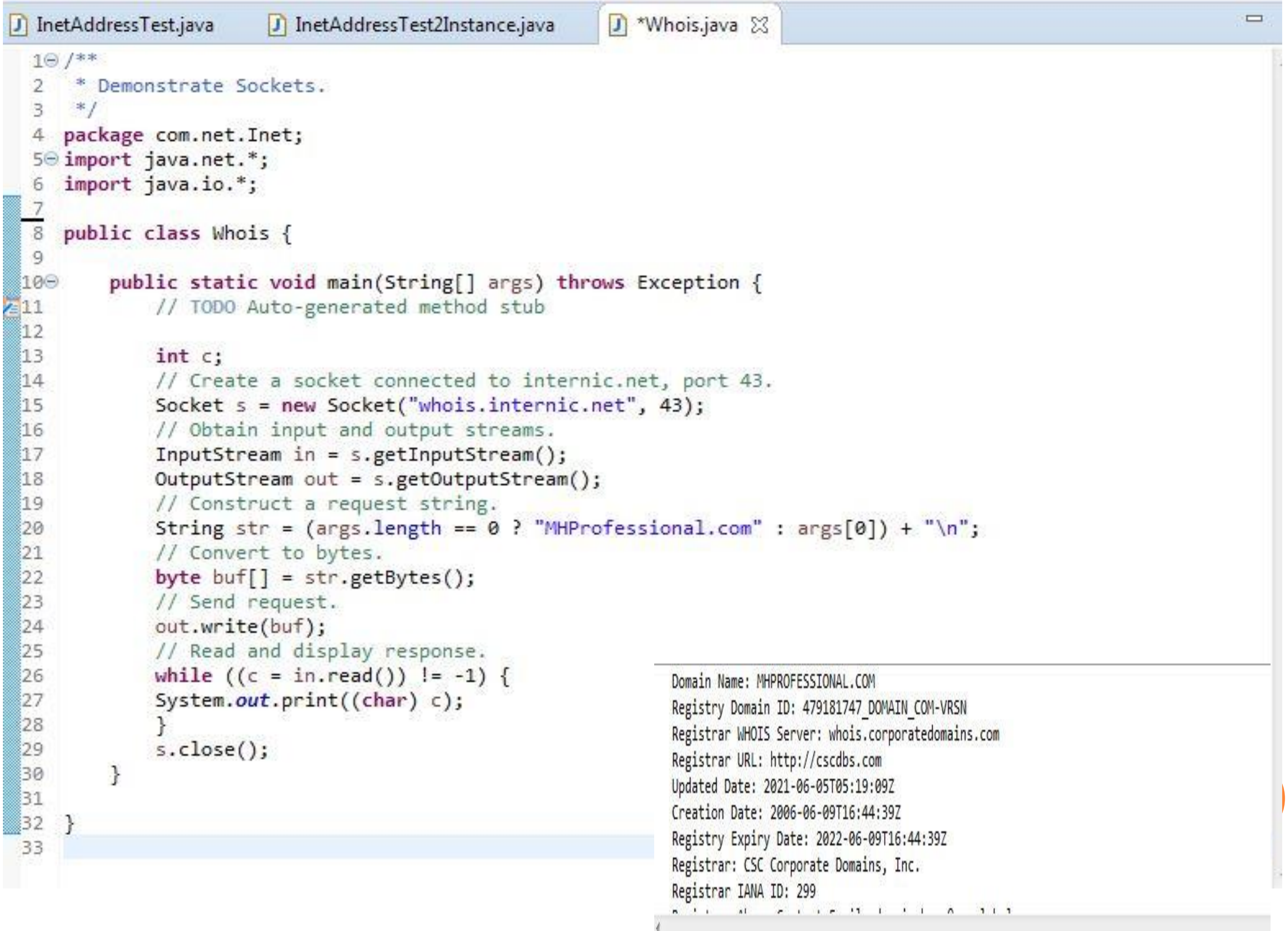
# TCP/IP Client Sockets EXAMPLE

- The following program provides a simple **Socket** example.

- It opens a connection to a "whois" port (port 43) on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned.

- InterNIC will try to look up the argument as a registered Internet domain name, and then send back the IP address and contact information for that site.

# TCP/IP Client Sockets EXAMPLE



```java
/**
 * Demonstrate Sockets.
 */
package com.net.Inet;
import java.net.*;
import java.io.*;

public class Whois {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub

        int c;
        // Create a socket connected to internic.net, port 43.
        Socket s = new Socket("whois.internic.net", 43);
        // Obtain input and output streams.
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();
        // Construct a request string.
        String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
        // Convert to bytes.
        byte buf[] = str.getBytes();
        // Send request.
        out.write(buf);
        // Read and display response.
        while ((c = in.read()) != -1) {
        System.out.print((char) c);
        }
        s.close();
    }

}
```

```
Domain Name: MHPROFESSIONAL.COM
Registry Domain ID: 479181747_DOMAIN_COM-VRSN
Registrar WHOIS Server: whois.corporatedomains.com
Registrar URL: http://cscdbs.com
Updated Date: 2021-06-05T05:19:09Z
Creation Date: 2006-06-09T16:44:39Z
Registry Expiry Date: 2022-06-09T16:44:39Z
Registrar: CSC Corporate Domains, Inc.
Registrar IANA ID: 299
```

# TCP/IP Server Sockets

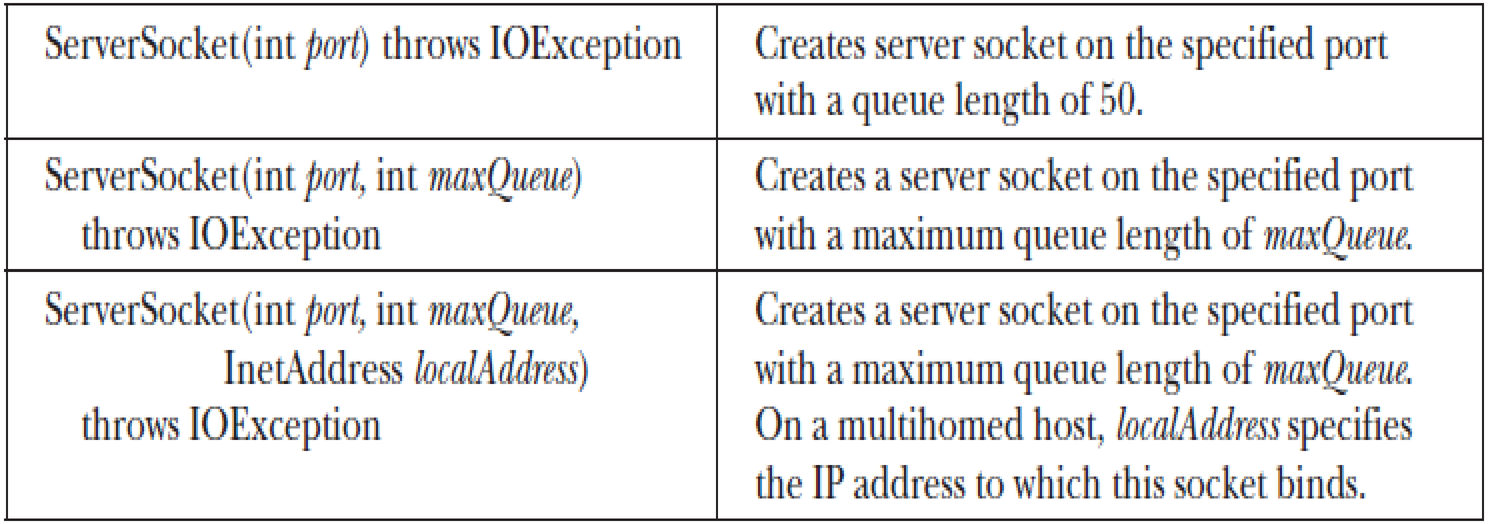
- Java has a different socket class that must be used for creating server applications.
- The **ServerSocket** class is used to create servers that listen for either local or remote client programs to connect to them on published ports. **ServerSocket**s are quite different from normal **Socket**s.
- When you create a **ServerSocket,** it will register itself with the system as having an interest in client connections.

# TCP/IP Server Sockets

- The constructors for **ServerSocket** reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be.

- The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. **The default is 50.**

- The constructors might throw an **IOException** under adverse conditions. Here are three of its constructors:



| | |
|---|---|
| ServerSocket(int *port*) throws IOException | Creates server socket on the specified port with a queue length of 50. |
| ServerSocket(int *port*, int *maxQueue*) throws IOException | Creates a server socket on the specified port with a maximum queue length of *maxQueue*. |
| ServerSocket(int *port*, int *maxQueue*, InetAddress *localAddress*) throws IOException | Creates a server socket on the specified port with a maximum queue length of *maxQueue*. On a multihomed host, *localAddress* specifies the IP address to which this socket binds. |

# Procedure for client

1. **Create the client socket for connection .**
2. **Acquire read & write streams to the socket for data transfer.**
3. **Use the stream according to the Server’s Protocols**
4. **Close the stream when conversation is over.**
5. **Close the client socket.**

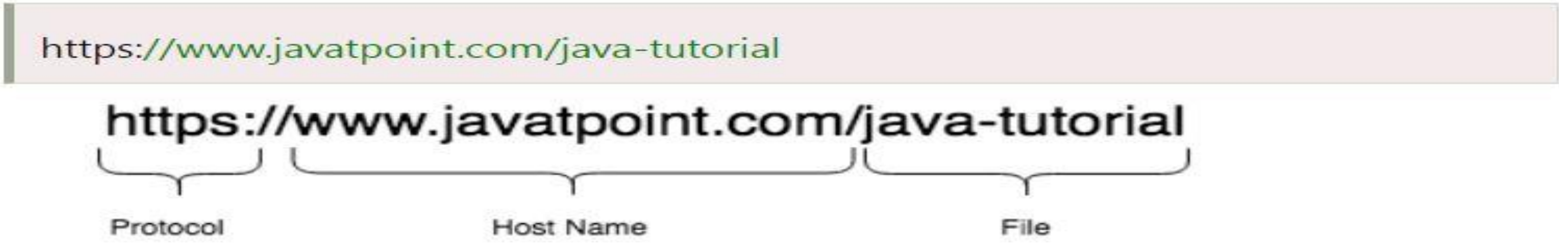
# Procedure for Server

1. **Create the Server Socket and begin Listening.**
2. **Call accept() method to get new connections from client sockets.**
3. **Create input and output streams for the return socket for data transfer.**
4. **Conduct the conversation based on agreed protocol.**
5. **Close the client streams and socket when the conversation is over.**
6. **Go back to step 4 to accept more client socket communication or go to step 7.**
7. **Close the server socket.**

# URL

- The **Java URL** class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web.

- For example:



https://www.javatpoint.com/java-tutorial

- A URL contains many information:

1. **Protocol:** In this case, http is the protocol.

2. **Server name or IP Address:** In this case, www.javatpoint.com is the server name.

3. **Port Number:** It is an optional attribute. If we write http//ww.javatpoint.com:80/sonoojaiswal/, 80 is the port number. If port number is not mentioned in the URL, it returns -1.

4. **File Name or directory name (Actual Path):** In this case, index.jsp is the file name.

# Constructors of URL class

- Java's **URL** class has several constructors; each can throw a **MalformedURLException**.
- One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

  **URL(String *urlSpecifier*) throws MalformedURLException**

- The next two forms of the constructor allow you to break up the URL into its component parts:

  **URL(String *protocolName*, String *hostName*, int *port*, String *path* )**
  **throws MalformedURLException**

  **URL(String *protocolName*, String *hostName*, String *path*)**
  **throws MalformedURLException**

# Constructors of URL class

- Another frequently used constructor allows you to use an existing URL as a reference context and then create a new URL from that context.

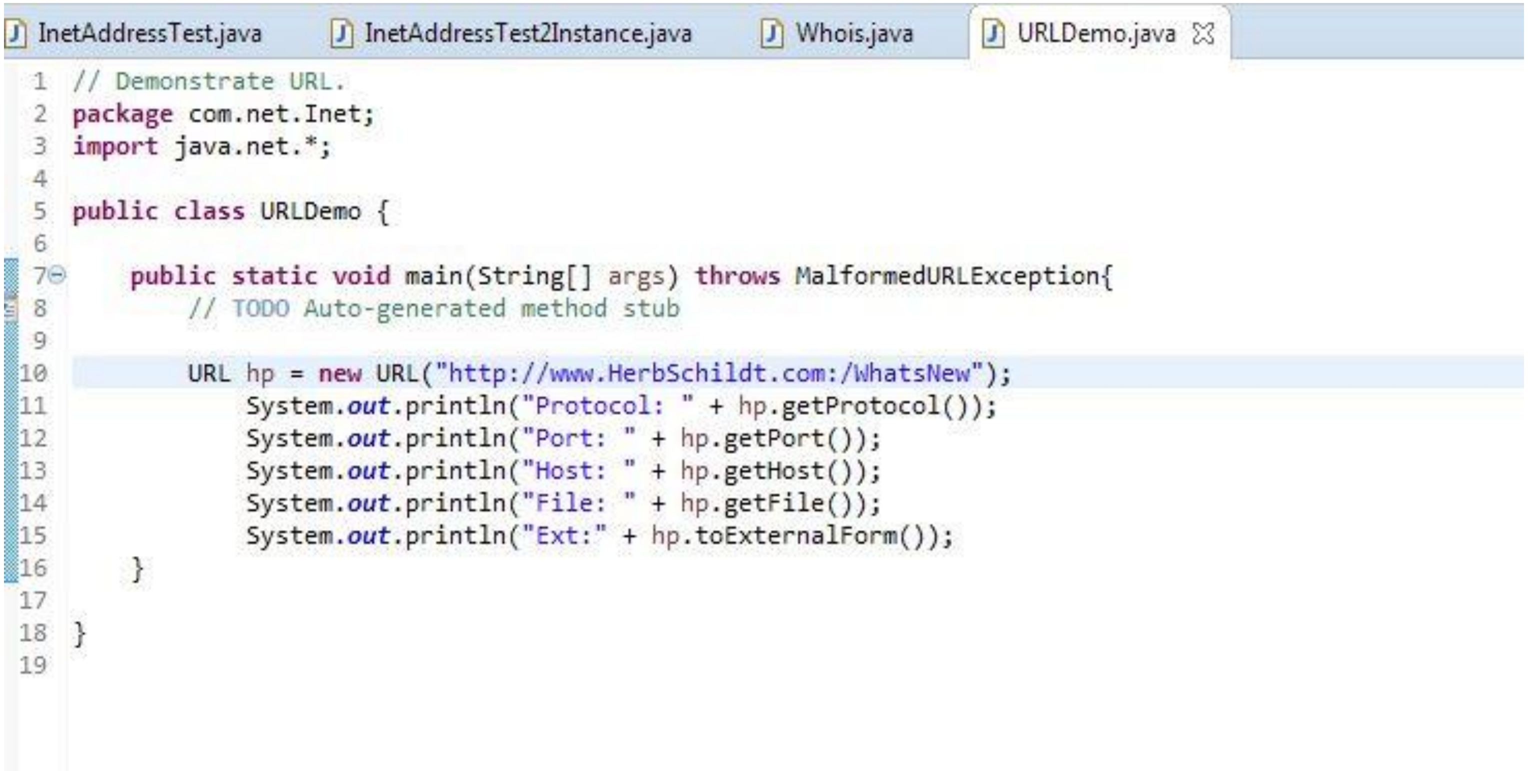
**URL(URL *urlObj*, String *urlSpecifier*) throws MalformedURLException**

- **COMMONLY USED METHODS OF URL CLASS:**

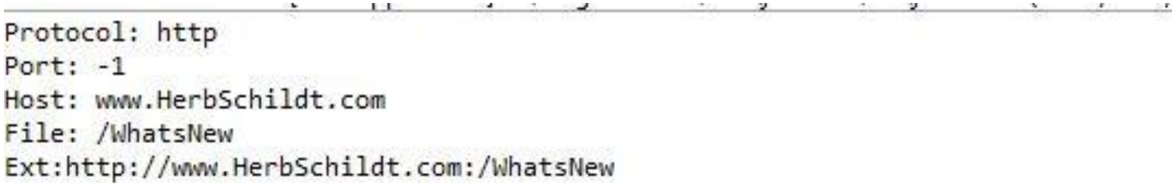
| Method Name | Description |
|---|---|
| public String getProtocol() | it returns the protocol of the URL. |
| public String getHost() | it returns the host name of the URL. |
| public String getPort() | it returns the Port Number of the URL. |
| public String getFile() | it returns the file name of the URL. |
| publicString toExternalForm() | it returns the string representation of a specified URL |

# URL class Example

The following example creates a URL to a page on **HerbSchildt.com** and then examines its properties:



```
// Demonstrate URL.
package com.net.Inet;
import java.net.*;

public class URLDemo {

    public static void main(String[] args) throws MalformedURLException{
        // TODO Auto-generated method stub

        URL hp = new URL("http://www.HerbSchildt.com:/WhatsNew");
            System.out.println("Protocol: " + hp.getProtocol());
            System.out.println("Port: " + hp.getPort());
            System.out.println("Host: " + hp.getHost());
            System.out.println("File: " + hp.getFile());
            System.out.println("Ext:" + hp.toExternalForm());
    }

}
```



```
Protocol: http
Port: -1
Host: www.HerbSchildt.com
File: /WhatsNew
Ext:http://www.HerbSchildt.com:/WhatsNew
```

# URLConnection CLASS

- Given a **URL** object, you can retrieve the data associated with it.

- To access the actual bits or content information of a **URL, **create a **URLConnection** object from it, using its **openConnection( )** method, like this:

  **urlc = url.openConnection()**

**openConnection( )** has the following general form:

**URLConnection openConnection( ) throws IOException**

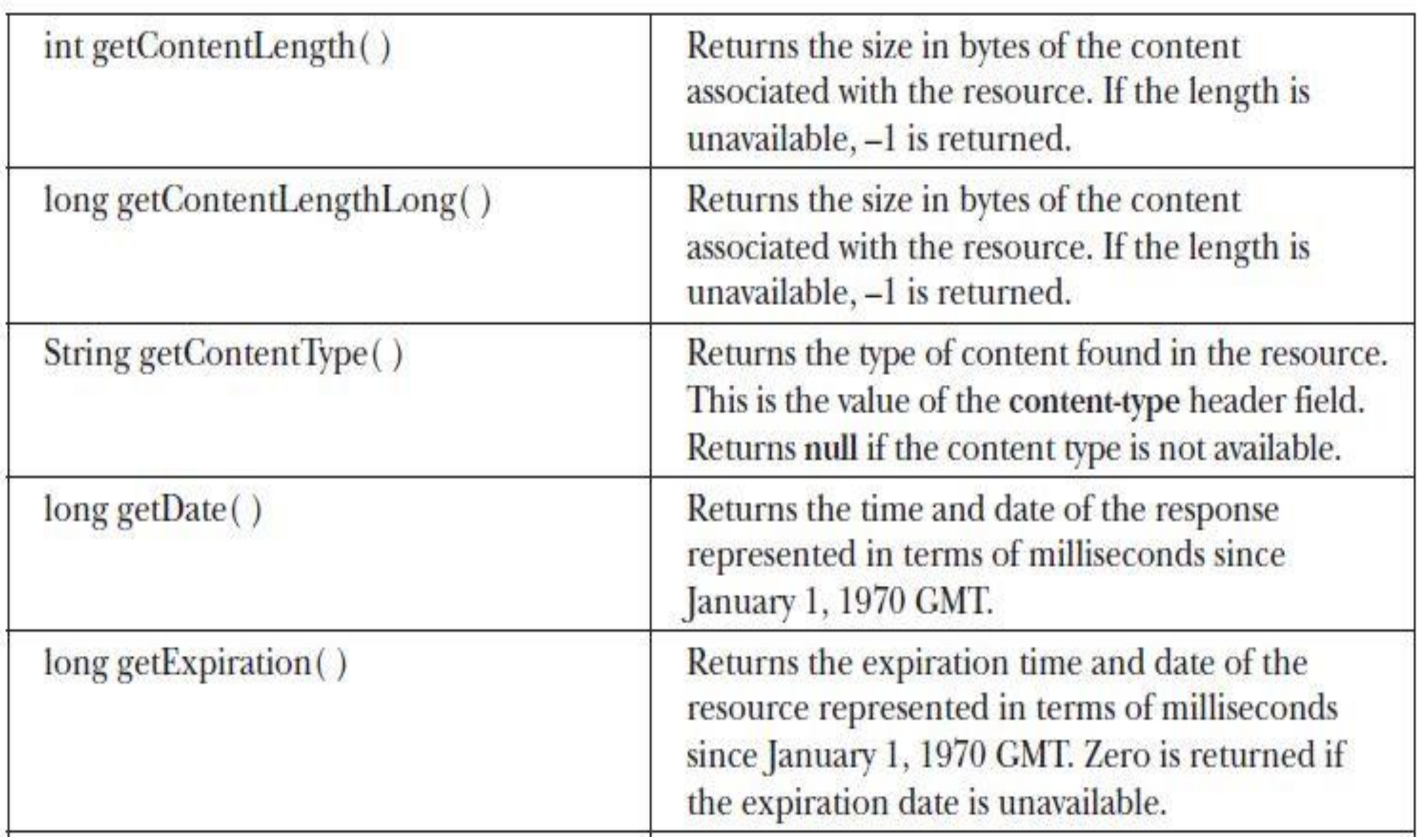
It returns a **URLConnection** object associated with the invoking **URL** object.

Notice that it may throw an **IOException**.

# URLConnection CLASS

- **URLConnection** is a general-purpose class for accessing the attributes of a remote resource.

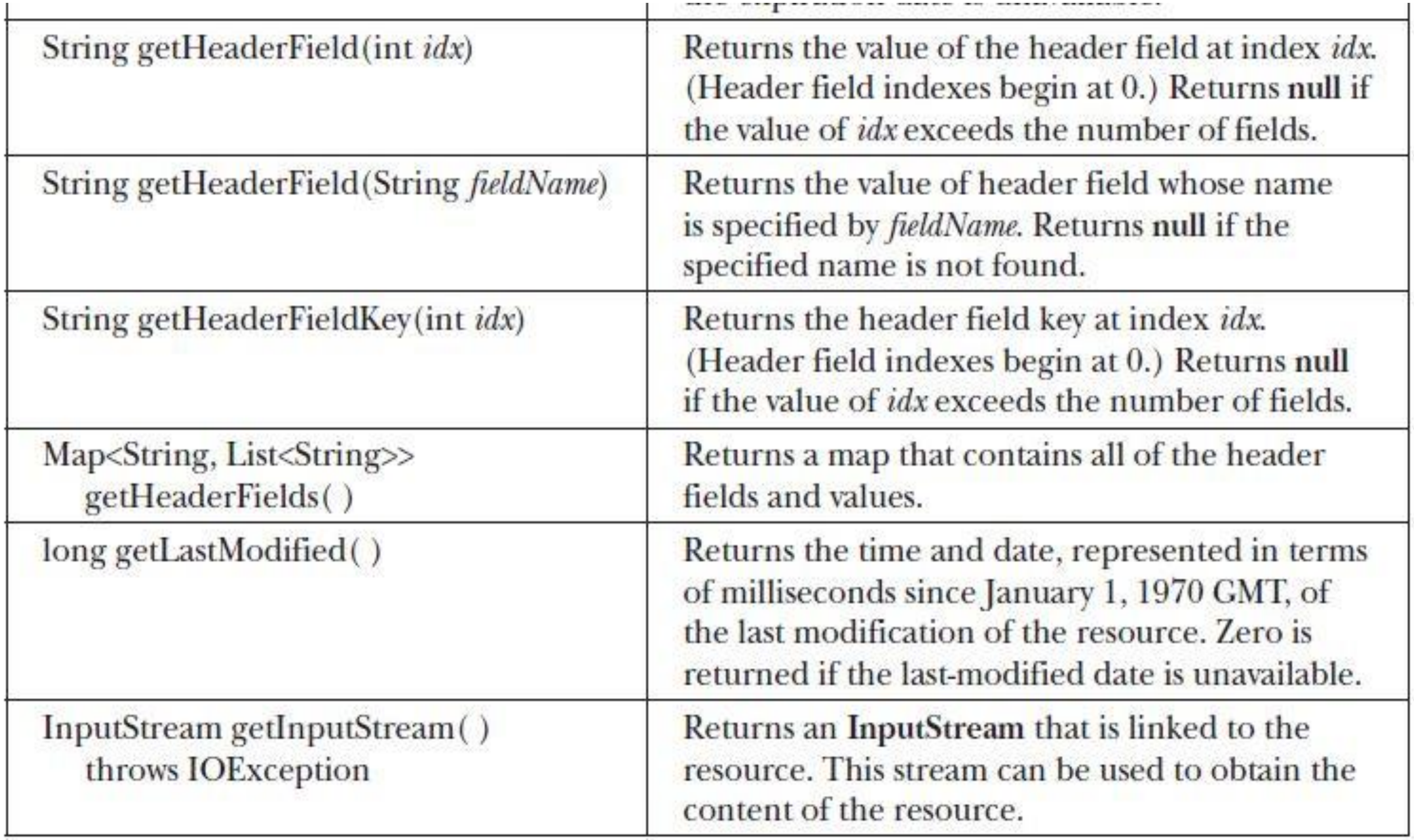
- **URLConnection** defines several methods. Here is a sampling:



| | |
|---|---|
| int getContentLength( ) | Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned. |
| long getContentLengthLong( ) | Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned. |
| String getContentType( ) | Returns the type of content found in the resource. This is the value of the **content-type** header field. Returns **null** if the content type is not available. |
| long getDate( ) | Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT. |
| long getExpiration( ) | Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable. |

# URLConnection CLASS

- **URLConnection** defines several methods. Here is a sampling:



| | |
|---|---|
| String getHeaderField(int *idx*) | Returns the value of the header field at index *idx*. (Header field indexes begin at 0.) Returns **null** if the value of *idx* exceeds the number of fields. |
| String getHeaderField(String *fieldName*) | Returns the value of header field whose name is specified by *fieldName*. Returns **null** if the specified name is not found. |
| String getHeaderFieldKey(int *idx*) | Returns the header field key at index *idx*. (Header field indexes begin at 0.) Returns **null** if the value of *idx* exceeds the number of fields. |
| Map<String, List<String>> getHeaderFields( ) | Returns a map that contains all of the header fields and values. |
| long getLastModified( ) | Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable. |
| InputStream getInputStream( ) throws IOException | Returns an **InputStream** that is linked to the resource. This stream can be used to obtain the content of the resource. |

# DATAGRAM SOCKET AND DATAGRAM PACKET

Java DatagramSocket and DatagramPacket classes are used for **connection-less socket programming**.

# DATAGRAMS

- ***Datagrams*** are bundles of information passed between machines.

- They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it.

- Likewise, when the datagram is received, there is no assurance that it hasn’t been damaged in transit or that whoever sent it is still there to receive a response.

- Java implements datagrams on top of the UDP protocol by using two classes:
  1) **DatagramPacket** object is the data container, while the

  2) **DatagramSocket** is the mechanism used to send or receive the **DatagramPackets**.

# DatagramSocket

- **DatagramSocket** defines four public constructors. They are shown here:
  1) DatagramSocket( ) throws SocketException
  2) DatagramSocket(int *port*) throws SocketException
  3) DatagramSocket(int *port*, InetAddress *ipAddress*) throws SocketException
  4) DatagramSocket(SocketAddress *address*) throws SocketException

- The first creates a **DatagramSocket** bound to any unused port on the local computer.
- The second creates a **DatagramSocket** bound to the port specified by *port*.
- The third constructs a **DatagramSocket** bound to the specified port and **InetAddress**.
- The fourth constructs a **DatagramSocket** bound to the specified **SocketAddress**

# DatagramSocket

- **DatagramSocket** defines many methods. Two of the most important are **send( )** and **receive( )**, which are shown here:

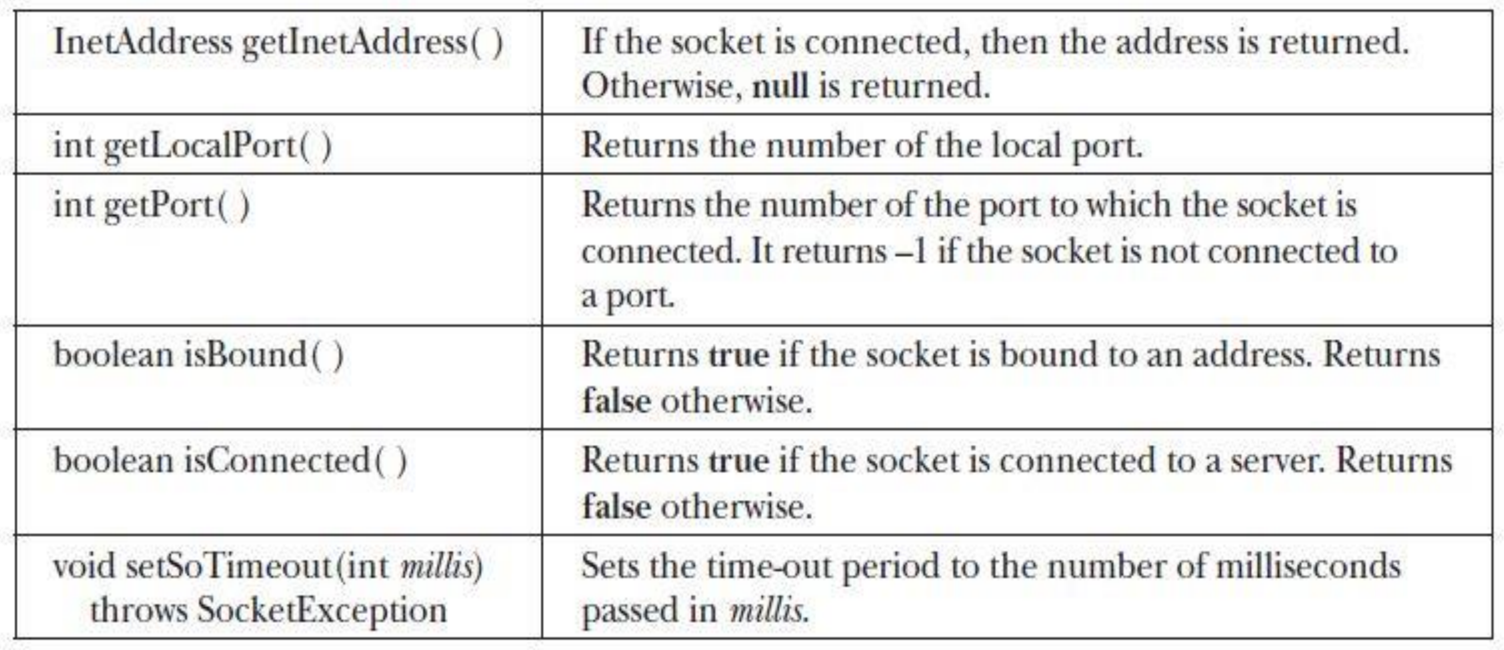
  1) void send(DatagramPacket *packet*) throws IOException
  2) void receive(DatagramPacket *packet*) throws IOException

- The **send( )** method sends a packet to the port specified by *packet*.
- The **receive( )** method waits for a packet to be received and returns the result.

- **DatagramSocket** also defines the **close( )**method, which closes the socket.

# DatagramSocket

- Other methods give you access to various attributes associated with a **DatagramSocket**.



| | |
|---|---|
| InetAddress getInetAddress( ) | If the socket is connected, then the address is returned. Otherwise, **null** is returned. |
| int getLocalPort( ) | Returns the number of the local port. |
| int getPort( ) | Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port. |
| boolean isBound( ) | Returns **true** if the socket is bound to an address. Returns **false** otherwise. |
| boolean isConnected( ) | Returns **true** if the socket is connected to a server. Returns **false** otherwise. |
| void setSoTimeout(int *millis*) throws SocketException | Sets the time-out period to the number of milliseconds passed in *millis*. |

# DatagramPacket

- **DatagramPacket** defines several constructors. Four are shown here:
  - DatagramPacket(byte *data* [ ], int *size*)
  - DatagramPacket(byte *data* [ ], int *offset*, int *size*)
  - DatagramPacket(byte *data* [ ], int *size*, InetAddress *ipAddress*, int *port*)
  - DatagramPacket(byte *data* [ ], int *offset*, int *size*, InetAddress *ipAddress*, int *port*)

- The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a **DatagramSocket.**

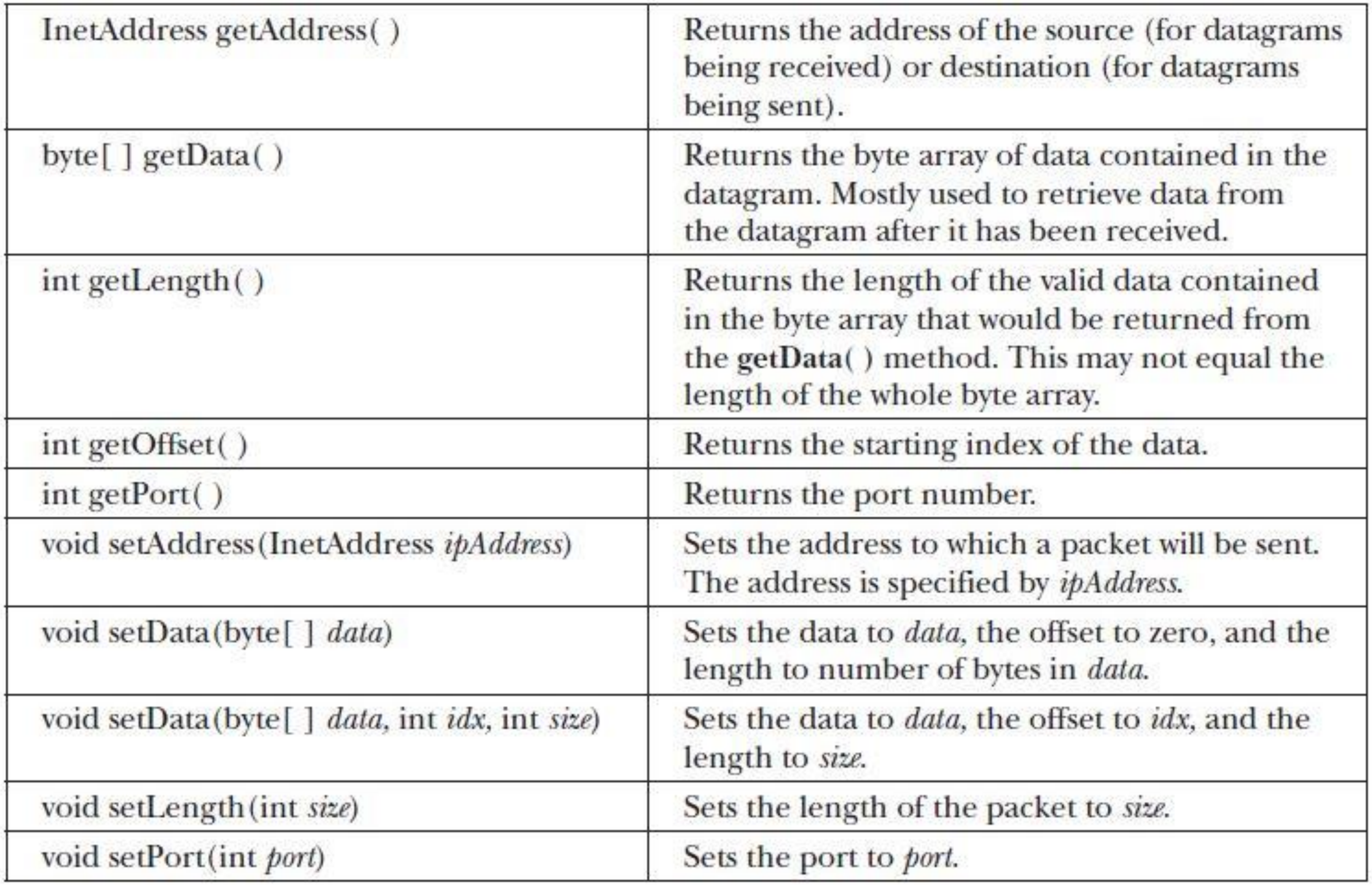
- The second form allows you to specify an offset into the buffer at which data will be stored.

- The third form specifies a target address and port, which are used by a **DatagramSocket** to determine where the data in the packet will be sent.

- The fourth form transmits packets beginning at the specified offset into the data.

# DatagramPacket

- **DatagramPacket** defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length



| | |
|---|---|
| InetAddress getAddress( ) | Returns the address of the source (for datagrams being received) or destination (for datagrams being sent). |
| byte[ ] getData( ) | Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received. |
| int getLength( ) | Returns the length of the valid data contained in the byte array that would be returned from the **getData**( ) method. This may not equal the length of the whole byte array. |
| int getOffset( ) | Returns the starting index of the data. |
| int getPort( ) | Returns the port number. |
| void setAddress(InetAddress *ipAddress*) | Sets the address to which a packet will be sent. The address is specified by *ipAddress*. |
| void setData(byte[ ] *data*) | Sets the data to *data*, the offset to zero, and the length to number of bytes in *data*. |
| void setData(byte[ ] *data*, int *idx*, int *size*) | Sets the data to *data*, the offset to *idx*, and the length to *size*. |
| void setLength(int *size*) | Sets the length of the packet to *size*. |
| void setPort(int *port*) | Sets the port to *port*. |

# Proxy Server

- Proxy servers are related to firewalls. If a firewall prevents hosts on a network from making direct connections to the outside world, a proxy server can act as a go-between.
- Thus, a machine that is prevented from connecting to the external network by a firewall would make a request for a web page from the local proxy server instead of requesting the web page directly from the remote web server.
- One of the security **advantages** of using a proxy server is that external hosts only find out about the proxy server.
- They do not learn the names and IP addresses of the internal machines, making it more difficult to hack into internal systems.