Chapter 1

Ports and Sockets

1. **Port**
   is a logical connection to a computer and is identified by a number in the range 1–65535. Each port may be dedicated to a particular server/service.
   // Port numbers in the range 1–1023 are normally set aside for the use of **specified standard services**, often referred to as 'well-known' services. For example, port 80 is normally used by Web servers.
   // Application programs wishing to use ports for non-standard services should use port numbers in the range of 1024–65535.
2. **Socket**
   is used to indicate one of the two end-points of a communication link between two processes.
   // When a client wishes to make connection to a server, it will create a socket at its end of the communication link.
   // Upon receiving the client's initial request (on a particular port number), the server will create a new socket at its end that will be dedicated to communication with that particular client.

Why Socket?

- In most applications, there are likely to be multiple clients wanting the same service at the same time.
  // A common example of this requirement is that of multiple browsers (quite possibly thousands of them) wanting Web pages from the same server.
- The server needs some way of distinguishing between clients and keeping their dialogues separate from each other. This is achieved via the use of sockets.

Streams.

// **I/O** in Java is built on streams.

- Input streams read data; Output streams write data. All Output Streams have the same basic methods to **write data** and all Input Streams use the same basic methods to **read data**.
- Filter Streams can be chained to either an input stream or an output stream. Filters can modify the data as it's read or written.
- Readers and Writers can be chained to input and output streams to allow programs to read and write text (i.e., characters) rather than bytes.

Output Stream

Java's basic output class is **java.io.OutputStream**:
public abstract class OutputStream

Output Stream Methods

1. **write(int b)**
   This method takes an integer from 0 to 255 as an argument and writes the corresponding byte to the output stream. This is the fundamental method of output stream.
2. **write(byte[] data)**
   This method writes the bytes from the specified array to the output stream.
3. **write(byte[] data, int off, int len)**
   This method writes len bytes from the specified byte array starting at offset "off" to this output stream.
4. **flush()**
   It flushes this output stream and forces any buffered output bytes to be written out.
5. **close()**
   It closes this output stream and releases any system resources associated with this stream.

Advantages of close() method:

- close() releases any resources associated with the stream, such as file handles or ports.
- If the stream derives from a network, closing the stream terminates the connection.
- Once an Output stream is closed, further writes to it throw IOException.
- If you do not close a stream in a long running program, it can leak file handles, network ports and other resources.

Why Flush Streams?

You should flush all streams immediately before you close them. Otherwise data left in the buffer when the stream is closed is lost. Failing to flush when you need to can lead to unpredictable, unrepeatable program hangs that are extremely hard to diagnose.

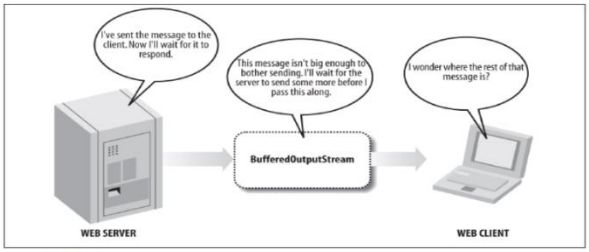


Figure 2-1. Data can get lost if you don't flush your streams

Input Stream

Java's basic output class is **java.io. InputStream**:
public abstract class InputStream

1. **read()**
   This method reads a single byte of data from the input stream's source and returns it as an int from 0 to 255. It is the basic method of input stream.
2. **read(byte[ ] data)**
   This method reads some number of bytes from the input stream and stores them into the specified array named data.
3. **read(byte[ ] data, int off, int len)**
   This method reads up to len bytes of data from the input stream beginning at offset "off" into the specified array named data.

Note// All the above three read() methods **return** –1 to signal the end of the stream.

4. **skip(long n)**
   On rare occasions, you may want to skip over data without reading it. The skip() method accomplishes this task.
5. **available()**
   You can use the available() method to determine how many bytes can be read without blocking. This returns the minimum number of bytes you can read. // On end of stream, available() **returns** 0.
6. **close()**
   This method closes this input stream and releases any system resources associated with the stream.

Filter Stream

// **InputStream** and **OutputStream** are fairly raw classes. They read and write bytes singly or in groups.
Java provides a number of filter classes you can attach to raw streams to translate the **raw bytes** to and from these and other formats.

The filters come in two versions:

1. The Filter Streams still work primarily with raw data as bytes: for instance, by compressing the data or interpreting it as binary numbers.
2. The Readers and Writers handle the special case of text in a variety of encodings such as **UTF-8** and **ISO 8859-1**.

// Every filter output stream has same write(), close(), and flush() methods as java.io.OutputStream.
// Every filter input stream has same read(), close(), and available() methods as java.io.InputStream.
// Filters are connected to streams by their constructor.

Predefined Streams

// **import java.lang** package. This package defines a class called System.

1. **System.out**
   It refers to the standard output stream. By default, this is the console.
2. **System.in**
   It refers to standard input, which is the keyboard by default.
3. **System.err**
   It refers to the standard error stream, which also is the console by default.

Note// **System.in** is an object of type InputStream; **System.out** and **System.err** are objects of type PrintStream. These are **byte streams**, even though they are typically used to read and write characters from and to the console.

Readers and Writers

- **java.io.Reader** class specifies the API by which characters are **read**.
- **java.io.Writer** class specifies the API by which characters are **written**.

// **readers** and **writers** use Unicode characters.
// The **FileReader** class and **FileWriter** class, work with the files.

Writer Class Methods

The Writer class mirrors the **java.io.OutputStream** class and it is abstract. It has five write() methods, a flush() and a close() method.

1. **write(int c)**
2. **write(char[] text)**
3. **write(String s)**
4. **write(char[] text, int offset, int length)**
5. **write(String s, int offset, int length)**
6. **flush()**
7. **close()**

OutputStreamWriter Class

An OutputStreamWriter receives characters from a Java program. It converts characters to bytes according to a specified encoding and writes them onto an underlying output stream.

Reader Class Methods

The Reader class mirrors the **java.io.InputStream** class. It is abstract.

1. **read()**
2. **read(char[] text)**
3. **read(char[] text, int offset, int length)**
4. **skip(long n)**
5. **ready()**
6. **close()**

Note// The ready( ) method returns a boolean indicating whether the reader may be read without blocking. It returns true if it is ready.

InputStreamReader & BufferedReader

- **InputStreamReader**: An InputStreamReader reads bytes from an underlying input stream. It converts bytes to characters according to a specified encoding and returns them.
- **BufferedReader**: When a program reads from a BufferedReader, text is taken from the buffer rather than directly from the underlying input stream or other text source. BufferedReader have the usual methods that are associated with reader.
- **Constructors of BufferedReader**:
  - public BufferedReader(Reader in, int bufferSize)
  - public BufferedReader(Reader in)

  Note// The BufferedReader class also has a readLine() method that reads a single line of text and returns it as a string.

PrintWriter

The PrintWriter class is a replacement for PrintStream class. We should use should use PrintWriter instead of PrintStream.The PrintWriter class has an almost identical collection of methods to PrintStream.

Constructors of PrintWriter:

- public PrintWriter(Writer out)
- public PrintWriter(Writer out, boolean autoFlush)
- public PrintWriter(OutputStream out)
- public PrintWriter(OutputStream out, boolean autoFlush)

Note// **PrintWriter** class has an almost identical collection of methods like PrintStream.

File Handling

// Class File is contained within package **java.io**

A text file requires a **FileReader** object for input and a **FileWriter** object for output.

File Objects are created first

- File **inFile** = new File("input.txt");
- File **outFile** = new File("output.txt");

Now, FileReader and FileWriter objects are created

- FileReader **in** = new FileReader(**inFile**);
- FileWriter **out** = new FileWriter(**outFile**);

Problem with FileReader and FileWriter

FileReader and FileWriter do not provide sufficient functionality or flexibility for reading and writing data from and to files.

To acquire this functionality, we need to:

- Wrap a **BufferedReader** object around a **FileReader** object in order to read from a file.
  BufferedReader input = new BufferedReader(new **FileReader**("in.txt"));
- Wrap a **PrintWriter** object around a **FileWriter** Object in order to write to the file.
  PrintWriter output = new PrintWriter(new **FileWriter**("out.txt"));

Note// Use methods readLine() from class BufferedReader and print(), println() from class PrintWriter.
When the processing of a file has been completed, the file should be closed via the close() method.
Closing the file causes the output buffer to be flushed and any data in the buffer to be written to disc.

File Methods

1. boolean **canRead()** - Returns true if file is readable and false otherwise.
2. boolean **canWrite()** - Returns true if file is writeable and false otherwise.
3. boolean **delete()** - Deletes file and returns true/false for success/failure.
4. boolean **exists()** - Returns true if file exists and false otherwise.
5. String **getName()** - Returns name of file.
6. boolean **isDirectory()** - Returns true if object is a directory/folder and false otherwise.
7. boolean **isFile()** - Returns true if object is a file and false otherwise.
8. long **length**() - Returns length of file in bytes.
9. String[] **list()** - If object is a directory, array holding names of files within directory is returned.
10. File[] **listFiles()** - Similar to previous method, but returns array of File objects.
11. boolean **CreateNewFile()** - Atomically creates a new, empty file if and only if a file with this name does not yet exist.
12. String **getParent()** - Returns the parent of the given file object.
13. String **getPath()** - Returns the path of the given file object.
14. String **getAbsolutePath()** - Returns the absolute pathname of the given file object.

Command Line Parameters

When entering the java command into a command window, it is possible to supply values in addition to the name of the program to be executed. These values are called **command line parameters** and are values that the program may make use of.

// Such values are received by method main as an array of Strings. If this argument is called arg ,then the elements may be referred to as arg[0] ,arg[1] , arg[2] , etc.

Chapter 2

InetAddress Class

InetAddress Class is in the package **java.net**. It handles Internet addresses both as host names and as IP addresses.

InetAddress Class Methods

1. **getByName()**
   This method uses DNS (Domain Name System) to return the Internet address of a specified host name as an InetAddress object.
2. **getLocalHost()**
   This method is used to retrieve the IP address of the current machine.

Getter Methods

1. **getHostName()** : This method returns a String that contains the name of the host with the IP address represented by this InetAddress object. If the machine doesn't have a hostname, a dotted quad format of the numeric IP address is returned.
2. **getCanonicalHostName()** : This method returns the fully qualified domain name for this IP address.
3. **getHostAddress()** : This method returns a string containing the dotted quad format of the IP address.
4. **getAddress()** : This method returns the IP address of a machine as an array of bytes in network byte order.

Server Socket

For servers that accept connections, Java provides a ServerSocket class that represents server sockets. A server socket runs on the server and listens for incoming TCP connections. Each server socket listens on a particular port on the server machine.

// When a client on a remote host attempts to connect to that port, the server wakes up, negotiates the connection between the client and the server, and returns a regular Socket object representing the socket between the two hosts.

Server sockets wait for connections while **client sockets initiate** connections.

// Once a ServerSocket has set up the connection, the server uses a regular Socket object to send data to the client.

Basic Life Cycle of a Server Program

1. A new **ServerSocket** is created on a particular port using a ServerSocket() constructor.
2. The ServerSocket listens for incoming connection attempts on that port using its accept() method. **accept()** blocks until a client attempts to make a connection at which point accept() returns a Socket object connecting the client and the server.
3. Depending on the type of server, either the Socket's **getInputStream()** method, **getOutputStream()** method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both **close** the connection.
6. The server returns to step 2 and waits for the next connection.

Constructing Server Sockets

There are four public ServerSocket constructors:

1. public **ServerSocket(int port)** throws BindException, IOException
2. public **ServerSocket(int port, int queueLength)** throws BindException, IOException
3. public **ServerSocket(int port, int queueLength, InetAddress bindAddress)** throws IOException
4. public **ServerSocket()** throws IOException

Note// In all three constructors, you can pass 0 for the port number so the system will select an available port for you. A port chosen by the system like this is sometimes called anonymous port because you don't know its number in advance (though you can find out after the port has been chosen).

Steps to Create TCP Server Socket

1. Create a ServerSocket object.
   Example: ServerSocket servSock = new ServerSocket(1234);
2. Put the server into a waiting state.
   Example: Socket link = servSock.accept();
3. Set up input and output streams.
   Example: BufferedReader in = new BufferedReader(new InputStreamReader(link.getInputStream()));
   Example: PrintWriter out = new PrintWriter(link.getOutputStream(),true);
4. Send and receive data.
   Example: out.println("Awaiting data…");
   String input = in.readLine();
5. Close the connection (after completion of the dialogue).
   Example: link.close();

TCP Client Socket

The **java.net.Socket** class is Java's fundamental class for performing client-side TCP operations.

Basic Constructors

1. public **Socket(String host, int port)** throws UnknownHostException, IOException
2. public **Socket(InetAddress host, int port)** throws IOException

Steps to Create TCP Client Socket

1. Establish a connection to the server.
   Example: Socket link = new Socket(InetAddress.getLocalHost(),1234);
2. Set up input and output streams.
   Example: BufferedReader in = new BufferedReader(new InputStreamReader(link.getInputStream()));
   Example: PrintWriter out = new PrintWriter(link.getOutputStream(),true);
3. Send and receive data.
   Example: out.println("Enter message");
   String input = in.readLine();
4. Close the connection.
   Example: link.close();

What is URL

A URL **(Uniform Resource Locator)** is a URI **(Uniform Resource Identifier)** that, as well as identifying a resource, provides a specific network location for the resource that a client can use to retrieve a representation of that resource.

//generic URI may tell you what a resource is, but not tell you where or how to get that resource.

The **java.net.URI** class only **identifies resources** and the **java.net.URL** class that can both **identify and retrieve resources.**

Syntax of URL: "protocol://userInfo@host:port/path?query#fragment"

- **protocol**: The protocol part can be file, ftp, http, https, telnet, or various other strings. Sometimes protocol is also called **scheme**.
- **host**: The host part of a URL is the name of the server that provides the resource you want.
- **userInfo**: The userInfo is optional login information for the server. If present, it contains a username and, rarely, a password.
- **port**: The port number is also optional. It's not necessary if the service is running on its default port (port 80 for HTTP servers).

Note// Together, the **userInfo**, **host**, and **port** constitute the **authority**.

- **path**: The path points to a particular resource on the specified server. As "/forum/index.php."
- **query**: The query string provides additional arguments for the server. It is commonly used only in **http** URLs, where it contains form data for input to programs running on the server.
- **fragment**: The fragment references a particular part of the remote resource. If the remote resource is HTML, the fragment identifier names an anchor in the HTML document.// If the remote resource is **XML**, the fragment identifier is an **XPointer**. Some sources refer to the fragment part of the **URL** as a "**section**". Java rather unaccountably refers to the **fragment** identifier as a "**Ref**".

URL Example

Example : URL http://www.ibiblio.org/javafaq/books/jnp/index.html? isbn=1565922069#toc,

1. the **scheme** (protocol) is **http**
2. the **authority** is **www.ibiblio.org**
3. the **path** is **/javafaq/books/jnp/index.html**
4. the **fragment** identifier is **toc**
5. the **query** string is **isbn=1565922069**.

URL Class

Read-only access to the parts of a URL is provided by ten public methods:

1. **get File()**

   The getFile() method returns a String that contains the path portion of a URL;
   For example: URL page = this.getDocumentBase();
   System.out.println("This page's path is " + page.getFile());
   If the URL does not have a file part, Java sets the file to the empty string.

2. **getHost()**

   The getHost() method returns a String containing the hostname of the URL.
   URL u = new URL("https://xkcd.com/727/");
   System.out.println(u.getHost());

3. **getPort()**

   The getPort() method returns the port number specified in the URL as an **int**. If no port was specified in the URL, getPort() **returns -1** to signify that the URL does not specify the port explicitly, and will use the default port.
   The following code prints -1 for the port number because it isn't specified in the URL:
   URL u = new URL("http://www.ncsa.illinois.edu/AboutUs/");
   System.out.println("The port part of " + u + " is " + u.getPort());

4. **getProtocol()**

   The getProtocol() method returns a String containing the scheme of the URL ("http" , "https" , or "file").
   URL u = new URL("https://xkcd.com/727/");
   System.out.println(u.getProtocol());

5. **getRef()**

   The getRef() method returns the fragment identifier part of the URL. If the URL doesn't have a fragment identifier, the method returns null.
   URL u = new URL("http://www.ibiblio.org/javafaq/javafaq.html#xtocid1902914");
   System.out.println("The fragment ID of " + u + " is " + u.getRef()); // returns xtocid1902914

6. **getQuery()**

   The getQuery() method returns the query string of the URL. If the URL doesn't have a query string, the method returns null.
   URL u = new URL("http://www.ibiblio.org/nywc/compositions.phtml?category=Piano"); System.out.println("The query string of " + u + " is " + u.getQuery()); //returns category=Piano

7. **getPath()**

   The getPath() method is a near synonym for getFile(); that is, it returns a String containing the path and file portion of a URL. However, unlike getFile(), it does not include the query string in the String it returns, just the path.
   Note// Both getPath() and getFile() return the full path and filename. The only difference is that getFile() also returns the query string and getPath() does not.

8. **getUserInfo()**

   Some URLs include usernames and occasionally even password information. This information comes after the scheme and before the host; an @ symbol delimits it.
   in the URL http://elharo@java.oreilly.com/, the user info is **elharo**.
   If the URL doesn't have any user info, getUserInfo() returns **null**.

9. **getAuthority()**

   Between the scheme and the path of a URL, you'll find the authority. In the most general case, the authority includes the user info, the host, and the port.
   For example, in the URL ftp://mp3:mp3@138.247.121.61:21000/c%3a/
   - the authority is mp3:mp3@138.247.121.61:21000, the user info is mp3:mp3, the host is 138.247.121.61, and the port is 21000.

   The getAuthority() method returns the authority as it exists in the URL, with or without the user info and port.

10. **getDefaultPort()**

    The getDefaultPort() method returns the default port used for this URL's protocol when none is specified in the URL. If no default port is defined for the protocol, then it **returns -1**.
    For example, if the URL is **http**://www.userfriendly.org/, getDefaultPort() **returns 80**;
    if the URL is **ftp**://ftp.userfriendly.org:8000/, getDefault Port() **returns 21**.

Chapter 3

// Java's implementation of UDP is split into two classes: DatagramPacket and DatagramSocket.

- **DatagramPacket** class **stuffs** bytes of data into UDP packets called datagrams and lets you **unstuff** datagrams that you receive.
- **DatagramSocket** sends as well as receives UDP datagrams.
  - To **send** data, you put the data in a **DatagramPacket** and send the packet using a **DatagramSocket**.
  - To **receive** data, you take a DatagramPacket object from a DatagramSocket and then inspect the contents of the packet.

  //In UDP, everything about datagram, including the address to which it is directed, is included in the packet itself; the socket only needs to know the local port on which to listen or send.

DatagramPacket

Constructors for sending datagrams

1. **DatagramPacket(byte[] data, int length, InetAddress destination, int port)**
2. **DatagramPacket(byte[] data, int offset, int length, InetAddress destination, int port)**
3. **DatagramPacket(byte[] data, int length, SocketAddress destination)**
4. **DatagramPacket(byte[] data, int offset, int length, SocketAddress destination)**

The Getter Methods

DatagramPacket has six methods that retrieve different parts of a datagram: the actual data plus several fields from its header.

// These methods are mostly used for datagrams received from the network.

1. **InetAddress getAddress()**
   getAddress() method returns an InetAddress object containing the address of the remote host.
2. **int getPort()**
   The getPort() method returns an integer specifying the remote port.
3. **SocketAddress getSocketAddress()**
   The getSocketAddress() method returns a SocketAddress object containing the IP address and port of the remote host.
4. **byte[] getData()**
   The getData() method returns a byte array containing the data from the datagram.
5. **int getLength()**
   The getLength() method returns the number of bytes of data in the datagram.
6. **int getOffset()**
   This method simply returns the point in the array returned by getData() where the data from the datagram begins.

The Setter Methods

Java also provides several methods for changing the data, remote address, and remote port after the datagram has been created.

1. **public void setData(byte[ ] data)**
   The setData() method changes the payload of the UDP datagram. 8 Setter Methods (cont..)
2. **public void setData(byte[] data, int offset, int length)**
   The setData() method provides an alternative approach to sending a large quantity of data.
3. **public void setAddress(InetAddress remote)**
   The setAddress() method changes the address a datagram packet is sent to.
4. **public void setPort(int port)**
   The setPort() method changes the port a datagram is addressed to.
5. **public void setAddress(SocketAddress remote)**
   The setSocketAddress() method changes the address and port a datagram packet is sent to.
6. **public void setLength(int length)**
   The setLength() method changes the number of bytes of data in the internal buffer, that are considered to be part of the datagram's data.

DatagramSocket Class

To send or receive a DatagramPacket, you must open a datagram socket. In Java, a datagram socket is created and accessed through the DatagramSocket class.

Note// All datagram sockets bind to a local port, on which they listen for incoming data and which they place in the header of outgoing datagrams.

DatagramSocket Constructors

1. **public DatagramSocket() throws SocketException**
   This constructor creates a socket that is bound to an anonymous port. Pick this constructor for a client that initiates a conversation with a server.
2. **public DatagramSocket(int port) throws SocketException**
   This constructor creates a socket that listens for incoming datagrams on a particular port, specified by the port argument. this constructor write server that listens on a well-known port.
3. **public DatagramSocket(int port, InetAddress interface) throws SocketException**
   This constructor is primarily used on multihomed hosts. It creates a socket that listens for incoming datagrams on a specific port and network interface.
4. **public DatagramSocket(SocketAddress interface) throws SocketException**
   This constructor is like the previous one except that the address and port are read from a SocketAddress.
5. **protected DatagramSocket(DatagramSocketImpl impl) throws SocketException**
   This constructor enables subclasses to provide their own implementation of the UDP protocol.

Sending and Receiving Datagrams

The primary task of the DatagramSocket class is to send and receive UDP datagrams. **One socket can both send and receive**. Indeed, it can send and receive to and from multiple hosts at the same time.

1. **void send(DatagramPacket dp) throws IOException**
   Once a DatagramPacket is created and a DatagramSocket is constructed, send the packet by passing it to the socket's send() method.
2. **void receive(DatagramPacket dp) throws IOException**
   receives a single UDP datagram from the network and stores it in the preexisting DatagramPacket object.
3. **void close()**
   Calling a DatagramSocket object's close() method frees the port occupied by that socket.
4. **int getLocalPort()**
   returns an int that represents the port on which the socket is listening. Use this method if you created a DatagramSocket with an anonymous port and want to find out what port the socket has been assigned.
5. **InetAddress getLocalAddress()**
   returns an InetAddress object that represents the local address to which the socket is bound.
6. **SocketAddress getLocalSocketAddress()**
   returns SocketAddress object that wraps the local interface and port of the socket.

Steps to Create UDP Server

1. **Create a DatagramSocket object**
   - DatagramSocket datagramSocket = new DatagramSocket(1234);
2. **Create a buffer for incoming datagrams**
   - byte[] buffer = new byte[256];
3. **Create a DatagramPacket object for the incoming datagrams**
   - DatagramPacket inPacket = new DatagramPacket(buffer, buffer.length);
4. **Accept an incoming datagram**
   - datagramSocket.receive(inPacket);
5. **Accept the sender's address and port from the packet**
   - InetAddress clientAddress = inPacket.getAddress();
   - int clientPort = inPacket.getPort();
6. **Retrieve the data from the buffer**
   - String message = new String(inPacket.getData(),0,inPacket.getLength());
7. **Create the response datagram**
   - DatagramPacket outPacket = new DatagramPacket(response.getBytes(), response.length(),clientAddress, clientPort);
8. **Send the response datagram**
   - datagramSocket.send(outPacket);
9. **Close the DatagramSocket**
   - datagramSocket.close();

Steps to Create UDP Client

1. **Create a DatagramSocket object**
   - DatagramSocket datagramSocket = new DatagramSocket();
2. **Create the outgoing datagram**
   - DatagramPacket outPacket = new DatagramPacket(message.getBytes(), message.length(), host, PORT);
3. **Send the datagram message**
   - datagramSocket.send(outPacket);

Note// Steps 4–6 below are exactly the same as steps 2–4 of the server procedure.

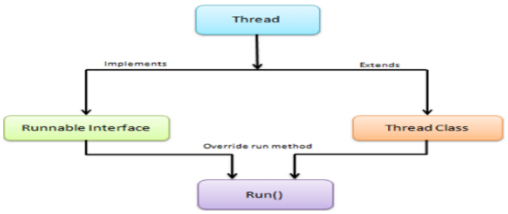
4. **Create a buffer for incoming datagrams**
   - byte[] buffer = new byte[256];
5. **Create a DatagramPacket object for the incoming datagrams**
   - DatagramPacket inPacket = new DatagramPacket(buffer,
6. **Accept an incoming datagram**
   - datagramSocket.receive(inPacket);
7. **Retrieve the data from the buffer**
   - buffer.length); String response = new String(inPacket.getData(),0, inPacket.getLength());
8. **Close the DatagramSocket**
   - datagramSocket.close();

Chapter 4

Thread Basics / Advantages

A thread is a flow of control through a program. A thread does **not** have a separate allocation of memory. A thread shares memory with other threads created by the same application. Threads created by an application can share global variables. Threads can have its own local variables. Each program will have at least one thread that is launched automatically by the JVM when that program is executed.

Using Threads in Java



You create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

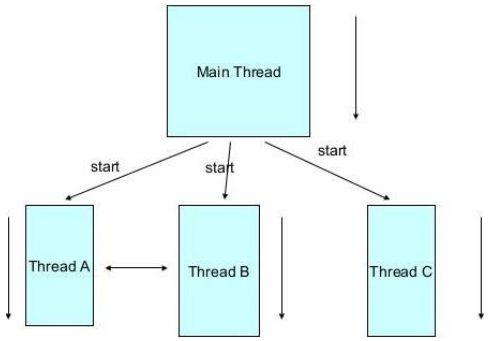
1. **Extending the Thread Class**
   - The **run** method specifies the actions that a thread is to execute and serves the same purpose for the process running on the thread as method **main** does for a full application program.
   - Like main , run may not be called directly. The containing program calls the start method (inherited from class Thread), which then automatically calls run.
   - The two most commonly used constructors are:
     1) **Thread()**
        If this constructor is used, the system generates a name of the form Thread-n, where n is an integer, starting at zero and increasing in value for further threads. Thus, if three threads are created via the first constructor, they will have names Thread-0 , Thread-1 and Thread-2 respectively.
     2) **Thread(String<name>)**
        This method provides a name for the thread via its argument.
   - **getName()** Whichever constructor is used, getName used to retrieve the name.
     Example:
     Thread firstThread = new Thread();
     System.out.println(firstThread.getName());
     Thread secondThread = new Thread("namedThread");
     System.out.println(secondThread.getName());
   - **sleep( )** This method is used to make a thread pause for a specified number of milliseconds.
     Example: myThread.sleep(1500); //Pause for 1.5 seconds.
   - **interrupt( )** This method may be used to interrupt an individual thread. In particular this method may be used by other threads to awaken a sleeping thread before that thread's sleeping time has expired.
2. **Implementing the Runnable Interface**
   - We first create an application class that explicitly implements the Runnable interface.
   - Then, in order to create a thread, we instantiate an object of our Runnable class and wrap it in a Thread object. We do this by creating a Thread object and passing the Runnable object as an argument to the Thread constructor.
   - There are two Thread constructors that allow us to do this.
     1) **Thread (Runnable<object>)**
     2) **Thread (Runnable<object>, String<name>)**

Thread Class Methods

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.
Interrupt	used by other threads to awaken a sleeping thread before that thread's sleeping time has expired.

Multithreading.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a **thread**, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.



Why use Multithreaded Server?

There is fundamental and important limitation associated with the server programs encountered so far.

1) They can handle only one connection at a time.
2) This restriction is simply not feasible for most real-world applications and would render the software useless.

Multithreaded Servers Advantages

1. It offers a **clean implementation**, by separating the task of allocating connections from that of processing each connection.
2. It is **robust**, since a problem with one connection will not affect other connections.
3. Enables you to write efficient programs that **make maximum use of the processing power** available in the system.
4. Helps you **reduce idle time** because another thread can run when one is waiting.

Multithreaded Server Process

1. The basic technique involves a two-stage process:
   - The main thread (the one running automatically in method main) allocates individual threads to incoming clients.
   - The thread allocated to each individual client then handles all subsequent interaction between that client and the server (via the thread's run method).
2. Since each thread is responsible for handling all further dialogue with its particular client, the main thread can forget about the client once a thread has been allocated to it.
3. The main thread waits for clients to make connection and allocate threads to them.

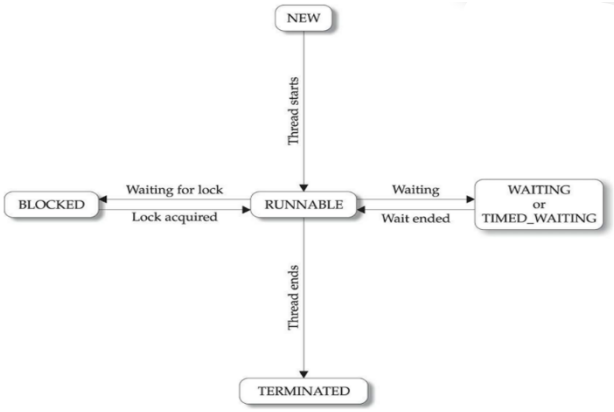
Thread States

You can obtain the current state of a thread by calling the getState( ) metho defined by Thread. For example, the following sequence determines if a thread called thrd is in the RUNNABLE state at the time getState( ) is called:

Thread.State ts = thrd.getState();

If(ts = = Thread. State.RUNNABLE)

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU
TERMINATED	A thread that has completed execution.
TIMED WAITING	A thread that has suspended execution for a specified period, such as when it has called sleep().
WAITING	A thread that has suspended execution because it is waiting for some action to occur.

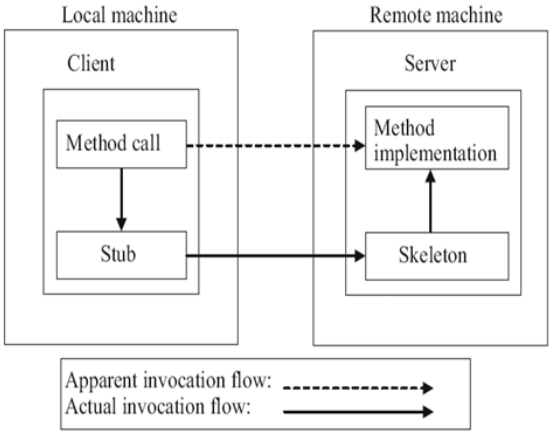


RMI Introduction

Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine.

Note// Once a reference to the remote object has been obtained, the methods of that object may be invoked in the same way as those of local objects.
// RMI will be making use of byte streams to transfer data and method invocations.
// In a distributed environment, it is often desirable to be able to invoke methods on remote objects (i.e., on objects located on other systems). RMI (Remote Method Invocation) provides a platform-independent means of doing this.

Basic RMI Process



1. The server program that has control of the remote object registers an **interface** with a naming service.
2. The interface contains the signatures for those methods of the object that the server wishes to make publicly available.
3. **Stub**: A stub is a local surrogate (a 'stand-in' or placeholder) for the remote object. It is at client side.
4. **Skeleton**: On the remote system, there will be another surrogate called a skeleton.
5. The client program invokes a method of the remote object. An equivalent method is being called in the stub. The stub then forwards the call and any parameters to the skeleton on the remote machine.
6. **Marshalling**: The serialising of parameters (only primitive types) is called marshalling .
7. **UnMarshalling**: The deserialisation of parameters is called unmarshalling.
8. Finally, the skeleton calls the implementation of the method on the server.

RMI Implementation

1. Create the interface.

This interface should import package **java.rmi** and must extend interface Remote, which (like Serializable) is a tagging interface that contains no methods.
The interface definition for this example must specify the signature for method **getGreeting**, which is to be made **available to clients**.
// This method must declare that it throws a **RemoteException**.

2. Define a class that implements this interface.

// The implementation file should import packages **java.rmi** and **java.rmi.server**.
The implementation class must extend class **RemoteObject** or one of RemoteObject 's subclasses.
// In practice, most implementations extend subclass **UnicastRemoteObject**, since this class supports **point-to-point** communication using **TCP streams**.
The implementation class must also implement our interface Hello, by providing an executable body for the single interface method **getGreeting**.
// In addition, we must provide a constructor for our implementation object. This constructor must declare that it throws a **RemoteException**.

3. Create the server process.

The server creates object(s) of the above implementation class and registers them with a naming service called the **registry**.
It does this by using static method **rebind** of class Naming (from package java.rmi). This method takes **two arguments**:

- A **String** that holds the **name** of the **remote object** as a URL with protocol rmi.
- A **reference** to the **remote object** (as an argument of type Remote).

The **rebind** method establishes a connection between the object's name and its reference.
Clients will then be able to use the remote object's name to retrieve a reference to that object via the registry.
Create the string URL holding the object's name. // The default **port** for **RMI** is **1099**.

4. Create the client process.

The client obtains a reference to the remote object from the **registry**.
// It does this by using method **lookup** of class Naming , supplying as an argument to this method the same URL that the server did when binding the object reference to the object's name in the registry.
// Since **lookup returns a Remote reference**, this reference must be typecast into an Hello reference (**not** an HelloImpl reference!).
Once the Hello reference has been obtained, it can be used to call the method that was made available in the interface.

Note// Please refer Textbook for the sample program.

RMI Compilation and Execution

1. Compile all files with javac.
   Note// that, before Java SE 5, it was necessary to compile the implementation class with the rmic compiler thus: rmic **HelloImpl** This would generate both a stub file and a skeleton file. However, this stage is no longer required.
2. Start the RMI registry.
3. Open a new window and run the server.
4. Open a third window and run the client.

RMI Security

1. The file java.policy define security restrictions. The file java.security defines the security properties.
2. Implementation of the security policy is controlled by an object of class RMISecurityManager (a subclass of **SecurityManager**).
3. We must create our own security manager that extends RMISecurityManager. This security manager must provide a definition for method checkPermission , which takes a single argument of class Permission from package java.security.

Example// import java.rmi.*; import java.security.*; public class ZeroSecurityManager extends RMISecurityManager { public void checkPermission(Permission permission) { System.out.println("checkPermission for : " + permission.toString()); } }

Multicast.

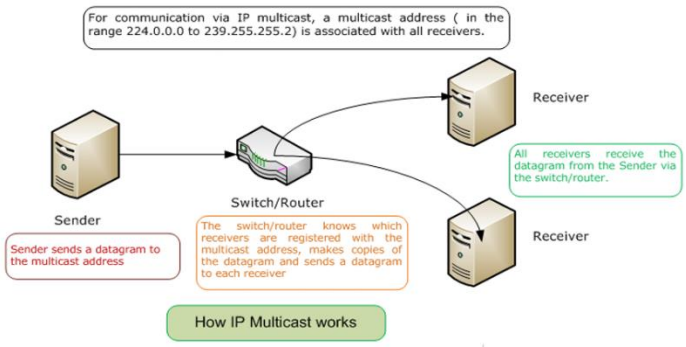
Multicasting sends data from one host to many different hosts, but **not** to everyone; the data only goes to clients that have expressed an interest by joining a particular multicast group.
// When a packet is multicast, it is addressed to a multicast group and sent to each host belonging to the group. It does not go to a single host (as in unicasting), nor does it go to every host (as in broadcasting).

Areas/Applications in Which Multicasting is Used

1. Audio and Video Applications.
2. Multiplayer Games.
3. Distributed File systems.
4. Massively Parallel Computing.
5. Multi person Conferencing.
6. Database Replication.
7. Content Delivery Networks.
8. Multicasting can be used to implement name services and directory services that don't require the client to know a server's address in advance.

How Multicast Works?

1. Multicasting has been designed to fit into the Internet as seamlessly as possible.
2. Most of the work is done by routers and should be transparent to application programmers.
3. An application simply sends datagram packets to a multicast address.
4. The routers make sure the packet is delivered to all the hosts in the multicast group.



Problem with Multicasting

The biggest problem is that multicast routers are not yet ubiquitous (everywhere); therefore, you need to know enough about them to find out whether multicasting is supported on your network.

Multicast Addresses

A multicast address is the shared address of a group of hosts called a multicast group.
// IANA **(Internet Assigned Numbers Authority)** is responsible for handing out permanent multicast addresses as needed.
The range of **IPv4 multicast addresses** is from 224.0.0.0 **to** 239.255.255.255. All addresses in this range have the binary digits 1110 as their first four bits.
IPv6 multicast addresses start with the byte 0xFF, or 11111111 in binary.
// A multicast address can also have a hostname. For example, the multicast address **224.0.1.1** (the address of the Network Time Protocol distributed service) is assigned the name ntp.mcast.net.
// Link-local multicast addresses begin with 224.0.0 (i.e., addresses from 224.0.0.0 to 224.0.0.255) and are reserved for routing protocols and other low-level activities, such as gateway discovery and group membership reporting.

Multicast Groups

A multicast group is a set of Internet hosts that share a multicast address. Any data sent to the multicast address is relayed to all the members of the group.

// **Membership** in a multicast group is **open**; hosts can **enter** or **leave** the group at **any time**.
// Groups can be either **permanent** or **transient**.

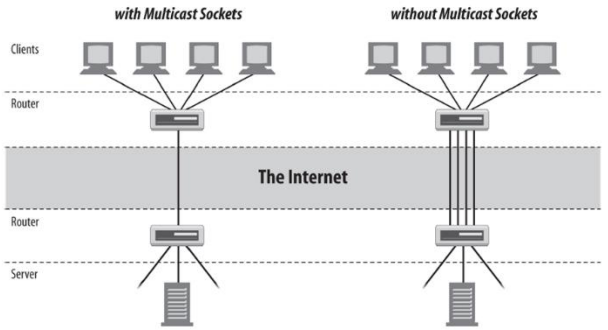
- **Permanent** groups have assigned addresses that remain constant, whether or not there are any members in the group.
- Most multicast groups are **transient** and exist only as long as they have members.

// All you have to do to create a new multicast group is pick a random address from 225.0.0.0 to 238.255.255.255, construct an InetAddress object for that address, and start sending it data.

Routers and Routing

a single server sending the same data to four clients served by the same router.

- A **multicast socket** sends one stream of data over the Internet to the clients' router; the router duplicates the stream and sends it to each of the clients.
- **Without multicast sockets**, the server would have to send four separate but identical streams of data to the router, which would route each stream to a client.



// The biggest restriction on multicasting is the availability of special multicast routers (**mrouters**). The mrouters are reconfigured Internet routers or workstations that support the IP multicast extensions.

Constructors - Multicast Socket

1. public **MulticastSocket()** throws SocketException
   MulticastSocket **ms1** = new MulticastSocket();
2. public **MulticastSocket(int port)** throws SocketException
   MulticastSocket **ms2** = new MulticastSocket(4000);
3. public **MulticastSocket(SocketAddress bindAddress)** throws IOException
   SocketAddress address = new InetSocketAddress("192.168.254.32", 4000);
   MulticastSocket **ms3** = new MulticastSocket(address);

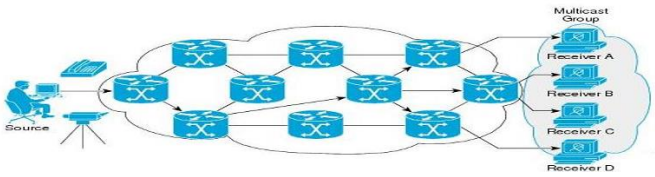
Note// All three constructors throw a **SocketException** if the Socket can't be created.

Communicating with a Multicast Group

Once a Multicast Socket has been created, it can perform four key operations:

1. **Joining Groups**
   To join a group, pass an InetAddress or a SocketAddress for the multicast group to the **joinGroup()** method.
   - void joinGroup(InetAddress address) throws IOException
   - void joinGroup(SocketAddress addr, NetworkInterface interface) throws IOException

   A single Multicast Socket can join **multiple multicast groups**.
   // Information about membership in multicast groups is stored in multicast routers.
2. **Sending Multicast Data**
   Sending data with a MulticastSocket is similar to sending data with a DatagramSocket. Stuff your data into a DatagramPacket object and send it off using the **send()** method. The data is sent to every host that belongs to the multicast group to which the packet is addressed.
3. **Receiving Data**
   Once you have joined a multicast group, you receive datagrams exactly as you receive unicast datagrams. You set up a DatagramPacket as buffer and pass it into this socket's **receive( )**.
4. **Leaving Group**
   Call the **leaveGroup()** method when you no longer want to receive datagrams from the specified multicast group, on either all or a specified network interface.
   - void leaveGroup(InetAddress address) throws IOException
   - void leaveGroup(SocketAddress Addr, NetworkInterface interface) throws IOException



Chapter 5

Secure Communications

// Confidential communication through an open channel such as the public Internet absolutely requires that data be encrypted. Most encryption schemes that lend themselves to computer implementation are based on the notion of a key.

- In traditional **secret key (or symmetric)** encryption, the same key is used to encrypt and decrypt the data. Both the sender and the receiver have to know the single key.
- In **public key (or asymmetric)** encryption, different keys are used to encrypt and decrypt the data. One key, called the public key, encrypts the data. This key can be given to anyone. A different key, called the private key, is used to decrypt the data. This must be kept secret but needs to be possessed by only one of the correspondents.

Java Secure Socket Extension

// In java, Secure Communication is implemented using JSSE **(Java Secure Socket Extension).**
JSSE allows you to create sockets that transparently handle the negotiations and encryption necessary for secure communication.
// JSSE shields you from the low-level details of how algorithms are negotiated, keys are exchanged, authenticated, and data is encrypted.

The Java Secure Socket Extension is divided into four packages:

1. **javax.net.ssl** The abstract classes that define Java's API for secure network communication.
2. **javax.net** abstract socket factory classes used instead of constructors to create secure sockets.
3. **java.security.cert** The classes for handling the public-key certificates needed for SSL.
4. **com.sun.net.ssl** The concrete classes that implement the encryption algorithms and protocols in Sun's reference implementation of the JSSE.

Creating Secure Client Sockets

1. You can create a socket using the createSocket() method of **javax.net.ssl.SSLSocketFactory**.
2. SSLSocketFactory is an abstract class that follows the abstract factory design pattern. You get an instance of it by invoking the static SSLSocketFactory.getDefault() method.
   SocketFactory factory = SSLSocketFactory.getDefault();
   Socket socket = factory.createSocket("login.ibiblio.org", 7000);

Once you have a reference to the factory, use one of these five overloaded **createSocket()** methods to build an SSLSocket.

- abstract **Socket createSocket(**String **host, int port)** throws IOException, UnknownHostException
- abstract **Socket createSocket(**InetAddress **host, int port)** throws IOException
- abstract **Socket createSocket(**String **host, int port,** InetAddress **interface, int localPort)** throws IOException, UnknownHostException
- abstract **Socket createSocket(**InetAddress **host, int port,** InetAddress **interface, int localPort)** throws IOException, UnknownHostException
- abstract **Socket createSocket(**Socket **proxy,** String **host, int port, boolean autoClose)** throws IOException

The first two methods create and return a socket that's connected to the specified host and port or throw an IOException if they can't connect.
The third and fourth methods connect and return a socket that's connected to the specified host and port from the specified local network interface and port.
The last createSocket() method begins with an existing Socket object that's connected to a proxy server. It returns a Socket that tunnels through this proxy server to the specified host and port. The autoClose argument determines whether the underlying proxy socket should be closed when this socket is closed. If autoClose is true, the underlying socket will be closed; if false, it won't be.

Choosing the Cipher Suites

- The getSupportedCipherSuites() method in SSLSocketFactory tells you which combination of algorithms is available on a given socket. // abstract **String[] getSupportedCipherSuites()**
- The getEnabledCipherSuites() method of SSLSocketFactory tells you which suites this socket is willing to use // abstract **String[] getEnabledCipherSuites()**
- You can change the suites the client attempts to use via the setEnabledCipherSuites() method // abstract void **setEnabledCipherSuites(String[] suites)**

Cipher Suite Example

protocol, **key exchange algorithm**, **encryption algorithm**, and **checksum**.

SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA

The above cipher suite means.

1. Secure Sockets Layer
2. Diffie-Hellman method for key agreement
3. no authentication
4. Data Encryption Standard encryption with 40-bit keys
5. Cipher Block Chaining, and the Secure Hash Algorithm checksum.

Session Management

SSL is commonly used on web servers, as Web connections tend to be transitory; every page requires a separate socket.

Because of the high overhead involved in handshaking between two hosts for secure communications, SSL allows sessions to be established that extend over multiple sockets.

Different sockets within the same session use the same set of public and private keys.

In the JSSE, sessions are represented by instances of the SSLSession interface;

getSession() of SSLSocket returns the Session this socket belongs to //abstract **SSLSession getSession()**

- To prevent a socket from creating a session that passes false to setEnableSessionCreation(), use // abstract **void setEnableSessionCreation(boolean allowSessions)**
- The getEnableSessionCreation() method returns true if multisocket sessions are allowed, false if they're not // abstract **boolean getEnableSessionCreation()**
- On rare occasions, you may even want to reauthenticate a connection The startHandshake() method does this // abstract **void startHandshake()** throws IOException

Client Mode

The setUseClientMode() method determines whether the socket needs to use authentication in its first handshake. It can be used for both client and server-side sockets.

- When **true** is passed in, it means socket is in client mode and will **not** offer to **authenticate itself**.
- When **false** is passed, it will try to **authenticate itself**.

// abstract **void setUseClientMode(boolean mode)** throws IllegalArgumentException

Note// This property can be set only once for any given socket. Attempting to set it a second time throws an IllegalArgumentException.

getUseClientMode() method simply tells you whether socket will use authentication in first handshake.

// abstract **boolean getUseClientMode()**

A secure socket on the server side (i.e., one returned by the accept() method of an SSLServerSocket) uses the setNeedClientAuth() method to require that all clients connecting to it authenticate themselves (or not) //abstract void **setNeedClientAuth(boolean nAuth)** throws IllegalArgumentException

The getNeedClientAuth() method returns true if the socket requires authentication from the client side, false otherwise // abstract **boolean getNeedClientAuth()**

Creating Secure Server Sockets

Secure Server Sockets are instances of javax.net.SSLServerSocket.// **class SSLServerSocket extends ServerSocket**

Like SSLSocket, all the constructors in this class are protected and instances are created by an abstract factory class, javax.net.SSLServerSocketFactory.// class **SSLServerSocketFactory extends ServerSocketFactory**

An instance of SSLServerSocketFactory is returned by a static **SSLServerSocketFactory.getDefault()**

// static **ServerSocketFactory getDefault()**

SSLServerSocketFactory has three overloaded create ServerSocket() methods

1. **ServerSocket createServerSocket(int port)** throws IOException
2. **ServerSocket createServerSocket(int port, int queueLength)** throws IOException
3. **ServerSocket createServerSocket(int port, int qLngth, InetAddress intrfce)**throws IOException

Configuring SSLServerSockets

SSLServerSocket provides methods to choose cipher suites, manage sessions, and establish whether clients are required to authenticate themselves.

Choosing the Cipher Suites // same as client methods page 16

Session Management

client and server must agree to establish a session. server uses setEnableSessionCreation() method to specify whether this will be allowed and getEnableSessionCreation() method to determine whether this is allowed.

Client Mode

setNeedClientAuth() method

- passing **true** specify that only connections in which client can authenticate itself will be accepted.
- passing **false**, you specify that authentication is not required of clients. The default is false.