TCP Sockets

But first ...

- Background on Threads

Multithreading

Overview of Multithreading in Java

- Java has build-in support for concurrent programming. This enables you to have multiple flows of control, represented by multiple threads within a single program (process).
  - A thread is a single sequential flow of control within a process.
  - Each thread has a separate execution path, with its own beginning, program flow, current point of execution, and end.
  - They are represented by **Thread** objects in Java.

Multithreading (Contd.)

- Multithreading allows a program to perform multiple tasks concurrently.
  - Although threads give the appearance of running concurrently, in a single processor system the interpreter is switching between the threads and running them one at a time.
  - Multiprocessor systems may actually run multiple threads concurrently.
  - The threads all run in the same memory space, i.e., they can all access the same memory and call methods as if they were in a normal single threaded process.

Why Threads?

- Threads are useful whenever you have multiple tasks that you want to run concurrently.
  - For example, consider a program that interactively reacts with the user, and that also needs to download a file over the networks.

The Thread Class

- You create and manipulate threads in Java using instances of the Thread class.
  - One way to use threads is to subclass Thread, and override its run() method.
  - The run() method is the heart of the Thread object. It should contain code to do whatever work the object needs to do.

```
class FileDownload extends Thread
{
    public void run()
    {
      // code to download file
    }
}
```

- The **Thread** class in defines in java.lang package.

Creating Threads

- Creating the thread is done by creating an instance of the new class.
  - You then invoke the **start()** method, inherited from **Thread**.
  - **start()** automatically invokes **run()**, and then **returns control immediately**.
  - **run()** will now execute in a separate thread.

```
FileDownload fd = new FileDownload();
fd.start();              // Returns control immediately
// Do whatever else you want to do
// e.g., continue to interact with user
```

- The new thread runs until its run() method finishes, or until you stop it explicitly.

The Runnable Interface

- You can also use classes that implement the **Runnable** interface to run code in separate threads.
  - The Runnable interface simply declares the run() method. It is also defined in the java.lang package.

```
class FileDownload implements Runnable
{
    public void run()
    {
       // Code to download file
    }
}
```

- To run the code is a separate thread, create an instance of Thread, with an instance of the **Runnable** as an argument to its constructor.
- Starting the thread with then cause the **Runnable**'s **run()** method to be executed.

```
Thread t = new Thread( new FileDownload() );
t.start();                    // Calls FileDownload's run() method
```

- This allows to have code executed in a thread without needing to subclass **Thread**. This is useful if you need to subclass a different class to get other functionality (e.g., the Applet class).

A Simple Example

```
// Define a class that extends Thread
class LoopThread extends Thread
{
    // Define its run() method to do what you want
    public void run()
    {
        for(int i=1; i <= 20; i++) System.out.println("I am loopy");
    }
}

public class Looper
{
    public static void main(String[] args)
    {
        // Create an instance of the thread
        LoopThread loopy = new LoopThread();

        // Start it up
        loopy.start();
    }
}
```

TCP Sockets

JAVA - Internet Addresses

- `java.net.InetAddress` class
- You get an address by using static methods:
- Create InetAddress object representing the local machine

```
InetAddress myAddress = InetAddress.getLocalHost();
```

- Create InetAddress object representing some remote machine

```
InetAddress ad = InetAddress.getByName(hostname);
```

JAVA - Printing Internet Addresses

- You get information from an InetAddress by using methods:

```
ad.getHostName();
ad.getHostAddress();
```

JAVA - The InetAddress Class

- Handles Internet addresses both as host names and as IP addresses
- Static Method getByName returns the IP address of a specified host name as an InetAddress object
- Methods for address/name conversion:

public static InetAddress getByName(String host) throws UnknownHostException
public static InetAddress[] getAllByName(String host) throws UnknownHostException
public static InetAddress getLocalHost() throws UnknownHostException

public boolean isMulticastAddress()
public String getHostName()
public byte[] getAddress()
public String getHostAddress()
public int hashCode()
public boolean equals(Object obj)
public String toString()

The Java.net.Socket Class

- Connection is accomplished through the constructors. Each Socket object is associated with exactly one remote host. To connect to a different host, you must create a new Socket object.

public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress address, int port) throws IOException
public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException
public Socket(InetAddress address, int port, InetAddress localAddress, int localPort) throws IOException

- **Sending and receiving data is accomplished with output and input streams. There are methods to get an input stream for a socket and an output stream for the socket**.

public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOException

- **There is a method to close a socket:**

public void close() throws IOException

The Java.net.ServerSocket Class

- The *java.net.ServerSocket* class represents a server socket. It is constructed on a particular port. Then it calls accept() to listen for incoming connections.
  - accept() blocks until a connection is detected.
  - Then accept() returns a java.net.Socket object that is used to perform the actual communication with the client.

public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException

public Socket accept() throws IOException [BLOCKING]
public void close() throws IOException

TCP Sockets

SERVER:

1. Create a ServerSocket object
   ServerSocket servSocket = new ServerSocket(1234);
2. Put the server into a waiting state
   Socket link = servSocket.accept(); //BLOCKING
3. Set up input and output streams
4. Send and receive data
   out.println(data);
   String input = in.readLine();
5. Close the connection
   link.close()

Set up input and output streams

- Once a socket has connected you send data to the server via an output stream. You receive data from the server via an input stream.
- Methods *getInputStream* and *getOutputStream* of class *Socket:*

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

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

TCP Sockets

CLIENT:

1. Establish a connection to the server
   Socket link =
   new Socket(inetAddress.getLocalHost(), 1234);
2. Set up input and output streams
3. Send and receive data
4. Close the connection

References

- Cisco Networking Academy Program (CCNA), Cisco Press.
- CSCI-5273 : Computer Networks, Dirk Grunwald, University of Colorado-Boulder
- CSCI-4220: Network Programming, Dave Hollinger, Rensselaer Polytechnic Institute.
- TCP/IP Illustrated, Volume 1, Stevens.
- Java Network Programming and Distributed Computing, Reilly & Reilly.
- Computer Networks: A Systems Approach, Peterson & Davie.
- http://www.firewall.cx
- http://www.javasoft.com