High Level Concurrency Objects and Sockets

All material not from online sources/textbook copyright © Travis Desell, 2012

Online References

High Level Concurrency Objects:

- http://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html

All About Sockets:

- http://docs.oracle.com/javase/tutorial/networking/sockets/index.html

Overview

- High Level Concurrency Objects
- Sockets
- Reading and Writing from Sockets
- Writing Server Sockets

High-Level Concurrency Objects

- Lock objects
- Executors
- Concurrent collections
- Atomic variables

Lock Objects

```
public boolean impendingBow(Friend bower) {
      Boolean myLock = false;
      Boolean yourLock = false;
      try {
          myLock = lock.tryLock();
          yourLock = bower.lock.tryLock();
      } finally {
          if (! (myLock && yourLock)) {
              if (myLock) {
                  lock.unlock();
              }
              if (yourLock) {
                  bower.lock.unlock();
              }
          }
      }
      return myLock && yourLock;
  }
```

- `Lock` objects support a `wait/notify` mechanism that can back out if the lock isn't immediately available or a timeout expires
- This can be used to solve deadlocks

Executors

- Sometimes you want first class control over thread management and creation
- What if you're running an application with 10,000+ concurrent objects? Most machines cannot allocate that many threads.

Executor Interfaces

The `java.util.concurrent` package has:

- `Executor` - base interface
- `ExecutorService` - subinterface of `Executor` which adds features that help manage tasks and their `Executor`
- `ScheduledExecutorService` - subinterface of `ExecutorService` supporting future and periodic execution of tasks

Executor

```
Runnable r = ...
//Run r in a thread
(new Thread(r)).start();

//Use an executor to run r
Executor e = ...
e.execute(r);
```

- An executor is almost identical to a thread (except it uses execute)
- However, it may not necessarily use a new thread to run r (it may use an already existing one) -- more on that in `ThreadPool`

ExecutorService

- `ExecutorServices` also have a `submit(...)` method
- submit can accept `Callable` objects and returns a `Future` value, which can be used to receive the result of the `Callable` object, and manage the status of `Callable` and `Runnable` objects

ScheduledExecutorService

- Adds a `schedule` method which executes a `Runnable` or `Callable` after a delay
- Can also specify `scheduleAtFixedRate` and `scheduleWithFixedDelay` to execute tasks repeatedly

Thread Pools

- Consist of a pool of worker threads to process tasks
- This allows many concurrent objects to be controlled with a fixed or somewhat fixed number of threads

Thread Pools 2

`java.util.concurrent.Executors` has many static methods to create Executors (or thread pools):

- `newFixedThreadPool` - a thread pool with a fixed number of threads
- `newCachedThreadPool` - a thread pool that creates new threads when needed but also reuses threads that have been released by their tasks
- `newSingleThreadPool` - a thread pool that uses only one thread
- and `ScheduledExecutorService` versions of the same

ExecutorService

```
package com.journaldev.threads;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class MyCallable implements Callable<String> {
     public String call() throws Exception {
        Thread.sleep(1000);
        //return the thread name executing this callable task
        return Thread.currentThread().getName();
    }

    public static void main(String args[]){
        //Get ExecutorService from Executors utility class, thread pool size is 10
        ExecutorService executor = Executors.newFixedThreadPool(10);
        //create a list to hold the Future object associated with Callable
        List<Future<String>> list = new ArrayList<Future<String>>();
        //Create MyCallable instance
        Callable<String> callable = new MyCallable();
        for(int i=0; i< 100; i++){
            //submit Callable tasks to be executed by thread pool
            Future<String> future = executor.submit(callable);
            //add Future to the list, we can get return value using Future
            list.add(future);
        }
        for(Future<String> fut : list){
            try {
                //print the return value of Future, notice the output delay in console
                // because Future.get() waits for task to get completed
                System.out.println(new Date() + "::" + fut.get());
            } catch (InterruptedException | ExecutionException e) {
                e.printStackTrace();
            }
        }
        //shut down the executor service now
        executor.shutdown();
    }
}
```

http://www.journaldev.com/1090/java-callable-future-example

Concurrent Collections

- BlockingQueue -- A queue that blocks if you add to a full queue, and blocks if you try to remove from an empty queue
- ConcurrentMap -- A Map with atomic methods
- ConcurrentNavigableMap -- a concurrent map that supports partial matches

Atomic Variables

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }
}
```

- Remember the counter class?
- This is faster and doesn't require using `synchronized`
- Java has many atomic class implementations
- http://java.sun.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html

Concurrent Random Numbers

```
int r = ThreadLocalRandom.current().nextInt(4, 77);
```

The ThreadLocalRandom class provides a way to get random numbers concurrently which is more efficient than Math.random() (which can be a bottleneck in a concurrent program).

The above code will get a random integer between 4 and 77.

http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadLocalRandom.html

Sockets

Sockets

Sockets are one end of a two-way connection between programs running on a network.

Programs use sockets to communicate (send data back and forth) over the Internet (or local networks).

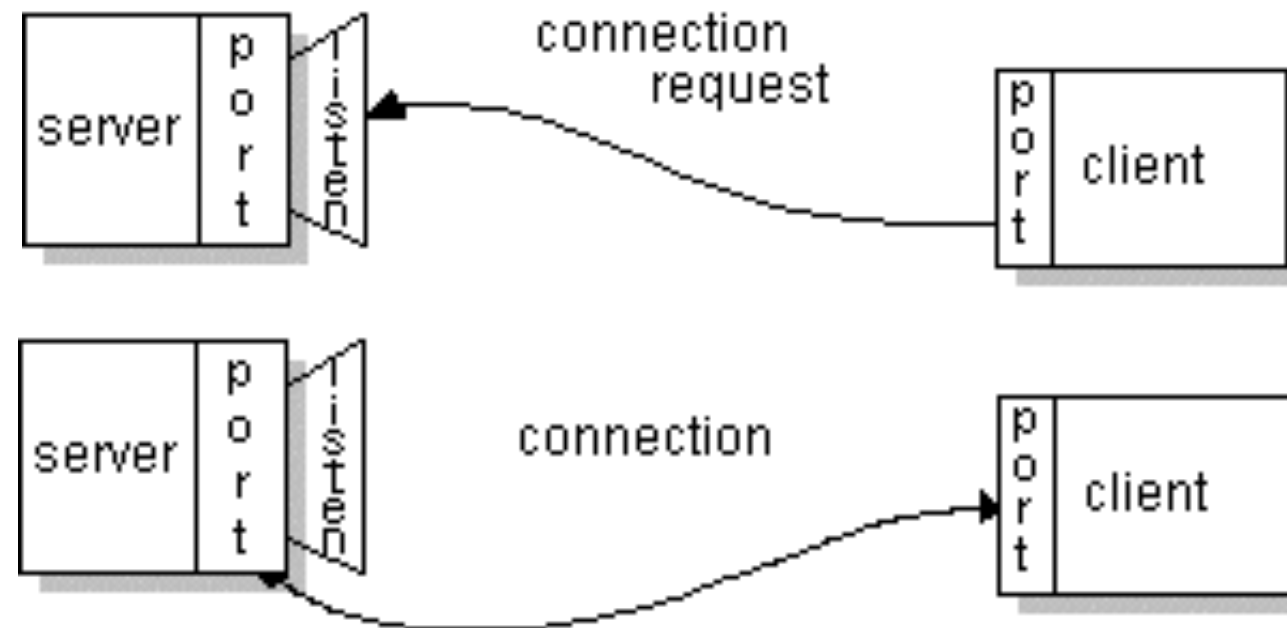
Sockets

Java provides two socket objects:

- java.net.Socket - which handles the client side connection

- java.net.ServerSocket - which handles the server side connection of the socket.

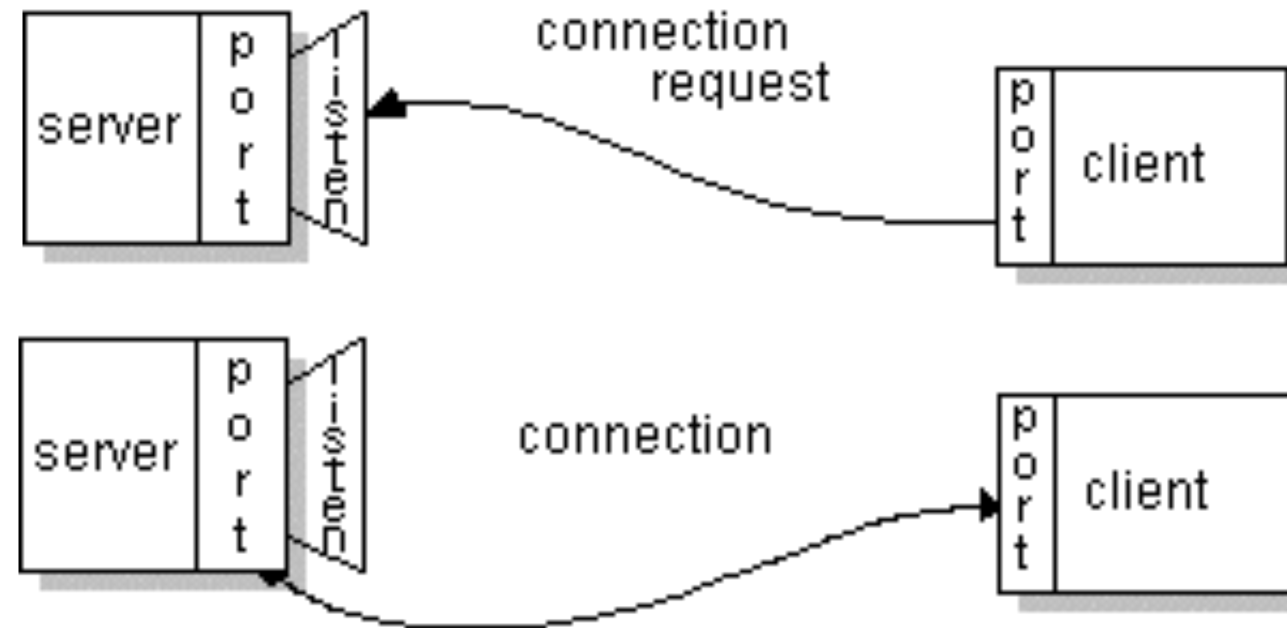
Sockets



The ServerSocket is *bound* to a port, which listens for incoming connections from clients.

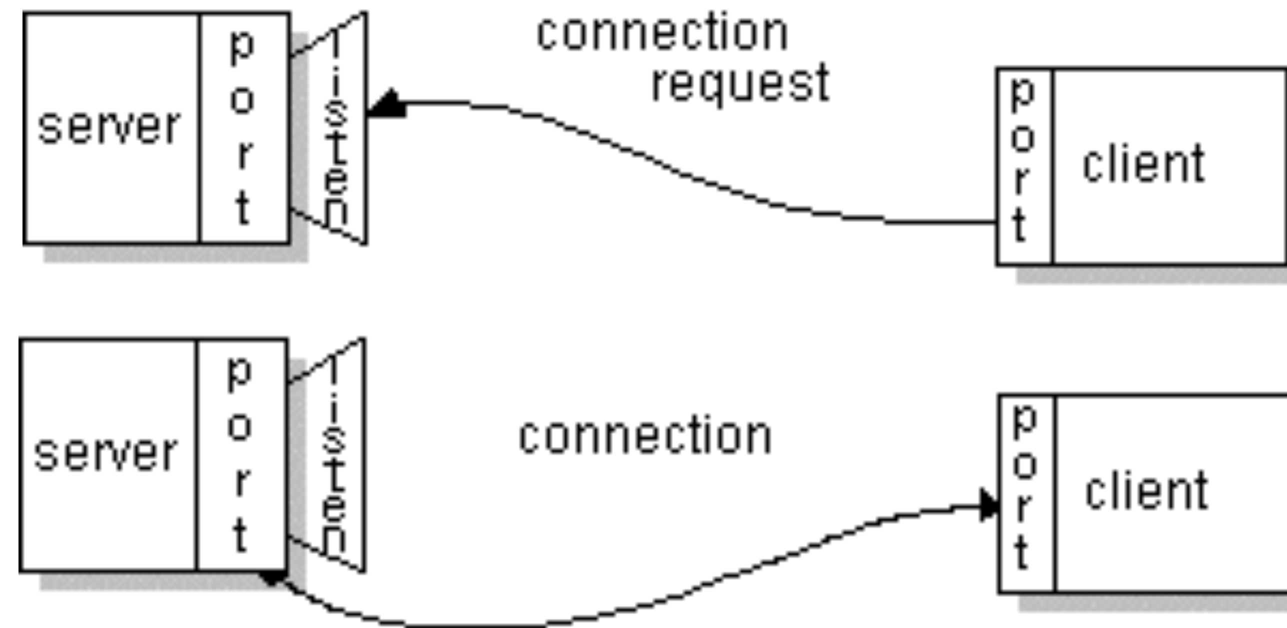
The client sockets connect to a hostname (like an ip address) and the port the ServerSocket is listening on.

Socket Definition



A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

Sockets



Java's Socket classes sit on top of native code, which allows the to communicate across *heterogeneous* systems (systems with different architectures and operating systems).

Web Connections

For connecting to the web (web pages) it is probably more appropriate to use Java's URLConnection and URLEncoder classes (more on that later):

http://docs.oracle.com/javase/tutorial/networking/urls/index.html

Reading and Writing from a Socket

EchoClient.java

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

public class EchoClient {
    public static void main(String[] args) throws IOException {

        if (args.length != 2) {
            System.err.println(
                "Usage: java EchoClient <host name> <port number>");
            System.exit(1);
        }

        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);

        try (
            Socket echoSocket = new Socket(hostName, portNumber);
            PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
            BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
            ) {
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput);
                System.out.println("echo: " + in.readLine());
            }
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                hostName);
            System.exit(1);
        }
    }
}
```

http://docs.oracle.com/javase/tutorial/networking/sockets/examples/EchoClient.java

Socket I/O

```
if (args.length != 2) {
    System.err.println(
        "Usage: java EchoClient <host name> <port number>");
    System.exit(1);
}

String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);
```

The EchoClient takes two arguments, the host name (which could also be an ip address) and the port number that the ServerSocket is listening on.

Socket I/O

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try {
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out = new PrintWriter(echoSocket.getOutputStream(),
true);

    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));

    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in));
```

The above code gives an example of how to connect to a ServerSocket with a client Socket.

Once you get a socket connection, you can treat it extremely similar to a file; using the same BufferedReader and BufferedWriter classes and methods.

Socket I/O

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try (
    Socket echoSocket = new Socket(hostName, portNumber);
    PrintWriter out = new PrintWriter(echoSocket.getOutputStream(),
true);

    BufferedReader in =
        new BufferedReader(
            new InputStreamReader(echoSocket.getInputStream()));

    BufferedReader stdIn =
        new BufferedReader(
            new InputStreamReader(System.in))
)
```

Note that the EchoClient gets a buffered reader both from System.in (what you type on the keyboard) and from the echoSocket (what the ServerSocket sends back).

Socket I/O

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

So what the echo client does after connecting is repeatedly read lines from standard input and then writes those lines to the server socket (which is the ‘out’ variable).

It then reads the line sent back from the server socket and prints it to the screen.

Writing Server Sockets

EchoServer.java

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

public class EchoServer {
    public static void main(String[] args) throws IOException {
       if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try (
            ServerSocket serverSocket =
                new ServerSocket(Integer.parseInt(args[0]));
            Socket clientSocket = serverSocket.accept();
            PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        ) {
            System.out.println("The server is listening at: " +
serverSocket.getInetAddress() + " on port " + serverSocket.getLocalPort());
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

http://docs.oracle.com/javase/tutorial/networking/sockets/examples/EchoServer.java

Writing a Server Socket

```
if (args.length != 1) {
     System.err.println("Usage: java EchoServer <port number>");
     System.exit(1);
 }

 int portNumber = Integer.parseInt(args[0]);
```

The Echo Server only takes one argument, the port it will listen on.

Writing a Server Socket

```
try (
    ServerSocket serverSocket =
        new ServerSocket(Integer.parseInt(args[0]));
    Socket clientSocket = serverSocket.accept();
    PrintWriter out =
        new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(clientSocket.getInputStream()));
) {
```

The ServerSocket gets created listening to the port, and then waits for an incoming client connection.

The accept method blocks until a client has made a connection. (Obvious question: how do you handle multiple clients? threads!)

Writing a Server Socket

```
try (
    ServerSocket serverSocket =
        new ServerSocket(Integer.parseInt(args[0]));
    Socket clientSocket = serverSocket.accept();
    PrintWriter out =
        new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(
        new InputStreamReader(clientSocket.getInputStream()));
) {
```

The ServerSocket creates streams for input and output similar to the client Socket.

Writing a Server Socket

```
            System.out.println("The server is listening at: " +
serverSocket.getInetAddress() + " on port " + serverSocket.getLocalPort());
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
```

The getInetAddress and getLocalPort methods can get the hostname/ip and the port a server socket is listening on.

Writing a Server Socket

```
            System.out.println("The server is listening at: " +
serverSocket.getInetAddress() + " on port " + serverSocket.getLocalPort());
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
```

The server socket repeatedly gets lines from the client, and then responds with the same line (thats why it’s called an “echo”).