CST8221 – Java Application Programming
Hybrid Activity #12
Interruptible Sockets

Terminology

As you already know, when a program connects to a socket, the current thread blocks until the connection has been established. Similarly, when a program reads and writes data through a socket, the current thread blocks until the operation is successful. The blocking of a thread could make the user GUI very irresponsive and annoying. One possible solution is to use socket and read/write timeout – see the lecture examples. Another solution is to use interruptible sockets. The Java 6 API provides such a solution.

The Nature of Things

In interactive GUI application you should give the user an option to cancel a socket connection that does not appear to work. However, if a thread blocks, you cannot unblock it by calling interrupt on the thread. If you want to interrupt a socket operation, you should use a *SocketChannel*, a class provided by the java.nio API package. Using this class you can open a socket like this:

```
SocketChannel channel = SocketChannel.open(new InetSocketAddress(host,port));
```

Unlike an ordinary socket, a channel does not have associated streams. Instead, it has *read()* and *write()* methods that make use of Buffer objects (see the Java documentation for more information about NIO buffers). These two methods are declared in interfaces *ReadableByteChannel* and *WritableByteChannel* correspondingly.

If you do not want to deal with buffers, you have the option to use the *Scanner* class to read from a *SocketChannel* object because the Scanner has a convenient constructor which takes a *ReadableByteChannel* type of parameter:

```
Scanner in = new Scanner(channel);
```

In order to turn a channel object into an output stream you can use a static method of the *Channels* class:

```
OutputStream out = Channels.newOutputStream(channels);
```

That is all you need to do. Whenever a thread is interrupted during an *open*, *read,* or *write* operation, the operation does not block, but it is terminated with an exception.

References

The Java API documentation.
“Core Java – Volume II – Advanced Features” by G.S. Horstman and G. Cornell, Prentice Hall

Code Example

You will find the code examples in ***CST8221_HA12_code_examples.zip***.
The code example demonstrates how to use the *interruptible sockets* to make a responsive Swing GUI.

Exercise

Run the **InterruptibleSocketTest** application. The program shows the difference between interruptible and blocking sockets. A server simulates blocking after sending 10 numbers to the client. After clicking on one of the Interruptible or Blocking button, a thread is started that connects to the server and prints the output. If you click the Cancel button during the printout of the first ten numbers, both threads will be interrupted.

Click the ***Blocking*** button first. Wait until the server stops counting to 10 and click on the ***Cancel*** button. Wait until the server thread closes – it will take for a while. Next, click the ***Interruptible*** button. Wait until the server stops counting to 10 and click on the ***Cancel*** button. The server thread will stop immediately.

Once you see how the example works, explore very carefully the code, and try to understand its inner workings. It could help you with your assignment. You do not need to use interruptible sockets in your implementation, but the example will help you to understand better how Java sockets work.

Questions

Q1. Does the thread establishing the connection block until the connection is established?
Q2. Can you interrupt a blocked thread?
Q3. Does a channel have associated streams?

Submission

No submission is required for this activity.

Marks

No marks are allocated for this activity, but remember that understanding how *the demo example works* is very important if you want to have a responsive networking GUI.

And do not forget that:

“To interrupt is rude but not if the opponent is a socket.” Networking joke