

Software Development (CS2500)

Lecture 43: Making a Connection

M.R.C. van Dongen

February 7, 2011

Contents

1	Outline	2
2	Making Connections	2
2.1	Bird's Eye View	2
2.2	Making the Connection	2
2.3	Reading from the Socket	3
2.4	Writing to the Socket	3
2.5	The Advisor	4
2.6	The Advisee	5
2.7	Running the Application	6
3	Threads	6
3.1	What are Threads?	7
3.2	Creating Threads	7
4	Race Conditions	8
4.1	Example	8
4.2	Avoiding Race Conditions	8
4.3	Synchronizing Methods	10
5	Deadlock	10
6	The Chat Application	10
7	For Wednesday	11
8	Acknowledgements	11

1 Outline

So far all our Java applications have used a single program. This lecture studies applications involving *multiple* programs. Each program involves a *process* which runs the program. We shall implement an application which lets two programs communicate. As we shall see we shall need several mini-processes within some programs: *threads*. As part of this we shall study the `Runnable` interface.

2 Making Connections

This lecture we shall implement a simple chat server application. The server program starts. Chatter programs can connect to the server. Chatters can send messages to the server. The server broadcasts all messages to its chatters. Upon receiving a message the chatters display it. To get started we shall implement a simple advice server.

Our advice server application involves an advisor and one advisee program. The advisor program starts. The advisee program requests a *connection*. The advisor establishes the connection. Next the advisor and advisee communicate. The communication involves writer and reader objects sitting on top of the connection. After establishing the connection the advisor sends some advice, and closes the connection.

2.1 Bird's Eye View

The server and client communication depends on the the server's IP address. There are three ingredients to communications.

Connect: The client and server establish a `Socket` connection. This is done as follows. The client requests the connection to the server's IP address at some *TCP port*. The server accepts the connection.

Send: The server sends a message. This is done by writing to a `PrintWriter` object.

Receive: The client receiver a message. This is done by reading from a `BufferedReader` object.

In general, writing to the server and reading from the client is also possible.

2.2 Making the Socket Connection

The socket connection is established by creating a `Socket` object. Creating the object formalises establishing the connection. After creating the connection both sides of the connection are aware of each other. The `Socket` class gives them communication for free.

The following shows how to create the `Socket` object:

```
Socket chatsocket = new Socket( <IP address>, <port> );
```

Java

The `<IP address>` is a `String`, e.g. "196.164.1.103". It uniquely identifies the server's machine. The `<port>` is an `int` representing the TCP port on the server's machine. The TCP port uniquely identifies some service on the server. For example, telnet runs on Port 23, ftp on Port 20, Valid ports are 0–65535. Ports 0–1023 are reserved.

2.3 Reading from the Socket

Reading from the Socket is easy. It involves a few steps.

1. Turn the Socket into an InputStream:

```
InputStream is = socket.getInputStream( );
```

Java

2. Turn the InputStream into an InputStreamReader:

```
InputStreamReader isr = new InputStreamReader( is );
```

Java

3. Turn the InputStreamReader into a BufferedReader:

```
BufferedReader reader = new BufferedReader( isr );
```

Java

4. Read:

```
String string = reader.readLine( );
```

Java

2.4 Writing to the Socket

Writing to the Socket works in a similar way as reading from the socket. It involves the following steps.

1. Turn the Socket into an OutputStream:

```
OutputStream os = socket.getOutputStream( );
```

Java

2. Turn the OutputStream into a PrintWriter:

```
PrintWriter writer = new PrintWriter( os );
```

Java

3. Write:

```
writer.println( "Hello world" );
```

Java

The `PrintWriter` is a *buffered* writer. Text written to the `PrintWriter` may not be written immediately. The text may still be in the buffer. *Flushing* the buffer empties the buffer and send it to the client.

```
writer.flush( );
```

Java

2.5 The Advisor

Remember that our application involves two programs: an advice server program, and an advisee program. The following demonstrates how the advisor program looks. For sake of this example, the program generates random advice.

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

public class AdviceServer {
    public static final int SOCKET = 5000;
    private static final Random rand = new Random( );
    private static final String[] advices = { "Go for it.", "Don't." };

    public static void main( String[] args ) {
        try {
            ServerSocket serverSocket = new ServerSocket( SOCKET );
            while (true) {
                Socket socket = serverSocket.accept( );
                OutputStream os = socket.getOutputStream( );
                PrintWriter writer = new PrintWriter( os );
                String advice = getAdvice( );
                writer.println( advice );
                writer.close( );
                System.out.println( "Gave advice: " + advice );
            }
        } catch (Exception exception) {
            // Omitted.
        }
    }

    private static String getAdvice( ) {
        return advices[ rand.nextInt( advices.length ) ];
    }
}
```

The server starts by creating a `ServerSocket`, which waits for requests to come in. The `ServerSocket` performs some operation based on that request, and then possibly returns a result to the requester. The number 5000 is chosen to represent the advice service on the server machine. The server and client have to agree on this number.

Next the server will start an infinite loop which listens to the socket.¹ The call to `serverSocket.accept()` *blocks* until a client requests a connection. A call which *blocks* on some condition won't return until

¹A proper implementation should do something which is a bit more intelligent than an infinite loop. For the purpose of this lecture starting an infinite loop is fine because it simplifies the rest of the class, allowing us to focus on what really matters.

the condition is met. In our case the call to `serverSocket.accept()` blocks until a client requests a connection, so this means that the call won't return until some client requests a connection.

2.6 The Advisee

The following is the Advisee class.

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

public class Advisee {
    private static final IP_ADDRESS = "127.0.1.1";

    public static void main( String[] args ) {
        try {
            Thread.sleep( 10000 );
            Socket socket = new Socket( IP_ADDRESS, AdviceServer.SOCKET );
            InputStream is = socket.getInputStream( );
            InputStreamReader isr = new InputStreamReader( is );
            BufferedReader reader = new BufferedReader( isr );
            for (int count = 0; count != 3; count++) {
                String advice = reader.readLine( );
                System.out.println( "Got advice: " + advice );
            }
            reader.close( );
        } catch (Exception exception) {
            // Omitted.
        }
    }
}
```

For this application we let the advisee sleep for 10 seconds. Letting the advisee sleep this long gives us enough time to get some information with the `ps` command in the Unix shell.

Upon waking up, the client continues by creating its `Socket` by connecting to Port 5000 on IP address 127.0.1.1. Notice that Port 5000 is the same as the one used by the server.

In our application the advisee reads three advices, prints them, and then closes the connection.

2.7 Running the Application

```
$ javac Advisee.java AdviceServer.java
$ java AdviceServer &
[1] 12442
$ java Advisee &
[2] 12456
$ ps -a
  PID TTY          TIME CMD
 12442 pts/1    00:00:00 java
 12456 pts/1    00:00:00 java
 12467 pts/1    00:00:00 ps
$ Gave advice: Go for it.
Got advice: Go for it.
Got advice: null
Got advice: null
# User hits return key and prompt appears
[2]+  Done                  java Advisee
$ ps -a
  PID TTY          TIME CMD
 12442 pts/1    00:00:00 java
 12478 pts/1    00:00:00 ps
$
```

Unix Session

Note that the advisee tries to get advice three times whereas the advice server only gives one advice. This shows why the last two advices are null. It is easy to let the advice server give more than one advice.

3 Threads

This section is an intermediate section which studies threads, which we need for our chat application. The following demonstrates why we need the threads. Our chatter programs do several things: they send messages, and they output incoming messages upon receiving them. How do we implement this? Let's assume we do it as follows:

```
while (! finished( )) {
    <write message>
    <read message>
    <display message>
}
```

Don't Try this at Home

This fails because the read may block. Swopping the read and write order doesn't change much.... We could use the `ready()` method to check if there's input. This would work, but it's not very pretty. It's much better if we could do the reading and writing at the same time.

3.1 What are Threads?

Threads are lightweight processes. One program can run several simultaneous threads. Threads live inside a process. They share the resources of the process. They have limited resources of their own. They have a small stack to enable function calls, and a small area with private data. Since they have limited resources, thread context switching is much faster than process context switching.

3.2 Creating Threads

The following demonstrates how to create a Thread.

```
public class ThreadExample implements Runnable {  
    private final int delay;  
    private final String name;  
  
    public static void main( String[] args ) {  
        Runnable first  = new ThreadExample( "first", 2 );  
        Runnable second = new ThreadExample( "second", 1 );  
        Thread firstThread = new Thread( first );  
        Thread secondThread = new Thread( second );  
        firstThread.start( );  
        secondThread.start( );  
    }  
  
    private ThreadExample( String name, int delay ) {  
        this.name = name;  
        this.delay = delay;  
    }  
  
    @Override  
    public void run( ) {  
        try {  
            Thread.sleep( delay * 1000 );  
        } catch( Exception exception ) {  
            // Omitted  
        }  
        System.out.println( name + " is done." );  
    }  
}
```

Even if you hadn't seen the `Runnable` interface before, you could tell, just by looking at the example, that it defines an abstract method `public void run()`. Without the `@Override` notation you might not have been able to tell this. This once more shows how important/useful it is to use the notation.

Note that the `main()` first runs `firstThread` and then runs `secondThread`. When we run the

application the following happens.

```
$ java threadExample
second is done.
first is done.
$
```

Unix Session

The output of the program may *seem* weird, but it is what we should expect. (Remember that this program is not a single-threaded sequential application.) The first Thread is started first. The Thread starts by running its `run()` method. The method runs concurrently with the Thread which is responsible for running the `main()`. This gives a grand total of two processes running at the moment. The two Threads are running concurrently but the first Thread goes to sleep for two seconds. While the first Thread is sleeping, the `main()` thread starts the second Thread. The second Thread goes to sleep for one second and wakes up *before* the first Thread. The rest is easy to explain

4 Race Conditions

A process has its own private address space. Threads don't: they share the *resources* of the process they are in. Resources can be: files, variables, and method access. Sharing resources violates encapsulation: it may lead to errors. To properly share their resources threads must respect resource dependencies. Dependencies are expressed as invariants. If the threads don't cooperate *race conditions* may occur. Here a *race condition* is a flaw whereby:

- The output/result of the program is ill-defined.
- The program depends on the right sequence or timing of other events.

Even read/write operations to/from memory may cause race conditions if they are not properly sequenced. (Note that race conditions may also occur if processes share resources such as files.)

4.1 Example

Figures 1 and 2 demonstrate how a race condition may occur. In both figures there are two threads which carry out the same statements. However, at the end of the two programs the "output" of the program (the values of the global variables) are different. What is more, an invariant which should seemingly hold is broken at the end of the sequence of event in Figure 2.

The cause of the race condition is that both threads have access to the global variables (shared resources) `v1`, `v2`, and `i`. Since they don't agree on how these shared resources should be used, their "cooperation," which should have maintained the invariant, fails.

4.2 Avoiding Race Conditions

Java has an easy solution to prevent (some) race conditions. The solution is to mark methods with the keyword `synchronized`. If a method is `synchronized` then at most one Thread can be "in" the method at the same time. (But it may call the method recursively.) When a method enters a `synchronized` method


```

// Shared resources
private int v1 = 0;
private int v2 = 0;
private int i = 0;

private
void f( int input ) {
    /* v1 == v2 */
    i = input;
    v1 += i;
    v2 += i;
    /* v1 == v2? */
}

```

Thread	Statement	v1	v2	i
—	—	0	0	0
1	<i>f(1)</i>	0	0	0
1	<i>i = input</i>	0	0	1
1	<i>v1 += i</i>	1	0	1
1	<i>v2 += i</i>	1	1	1
2	<i>f(2)</i>	1	1	1
2	<i>i = input</i>	1	1	2
2	<i>v1 += i</i>	3	1	2
2	<i>v2 += i</i>	3	3	2

Figure 1: Multi-threaded program. There are two threads: Thread 1 and Thread 2. Thread 1 carries out the call $f(1)$ and Thread 2 carries out the call $f(2)$. Both threads carry out the statements to the left. Their execution trace is listed in the table to the right. Each thread is supposed to maintain the invariant at the start and the end of the function $f()$. In this example, at most one thread is active at any moment in time. The number of the active process is listed in the column “Thread.” The statement which is carried out by the active thread is listed in the column “Statement.” The remaining three columns list the values of the variables immediately after the statement. In this example the switching of the threads is such that Thread 1 is finished before Thread 2 starts. After this sequence of events the invariant $v1 == v2$ still holds.

```

// Shared resources
private int v1 = 0;
private int v2 = 0;
private int i = 0;

private
void f( int input ) {
    /* v1 == v2 */
    i = input;
    v1 += i;
    v2 += i;
    /* v1 == v2? */
}

```

Thread	Statement	v1	v2	i
—	—	0	0	0
1	<i>f(1)</i>	0	0	0
1	<i>i = input</i>	0	0	1
1	<i>v1 += i</i>	1	0	1
2	<i>f(2)</i>	1	0	1
2	<i>i = input</i>	1	0	2
2	<i>v1 += i</i>	3	0	2
2	<i>v2 += i</i>	3	2	2
1	<i>v2 += i</i>	3	4	2

Figure 2: Multi-threaded program. There are two threads: Thread 1 and Thread 2. Thread 1 carries out the call $f(1)$ and Thread 2 carries out the call $f(2)$. Both carry out the statements to the left. Their execution trace is listed in the table to the right. Each thread is supposed to maintain the invariant at the end of the function $f()$. The number of the active process is listed in the column “Thread.” The statement which is carried out by the active thread is listed in the column “Statement.” The remaining three columns list the values of the variables immediately after the statement. This time the switching of the threads is interleaved. After this sequence of events the “invariant” $v1 == v2$ is broken.

this *locks* the method. If a method is locked, subsequent Thread calls will be blocked. Blocked Threads are put in a queue. When a method leaves a synchronized method this *unlocks* the method. When the method is unlocked some blocked Thread is awoken and allowed to enter the method.

4.3 Synchronizing Methods

The following demonstrates how to make a method synchronized.

```
<visibility modifier> synchronized <type> <name>( <argument list> ) {  
    <Do critical stuff>  
}
```

Java

5 Deadlock

Another problem with concurrent programs is *deadlock*. A program is deadlocked when two or more threads/processes are each waiting for each other to release a resource. Here resource may be a locked file, access to a printer, ..., or access to a synchronized method.

The following shows how two threads, Thread 1 and Thread 2, can create a deadlock situation.

- Both threads need two resources called a and b.
- For simplicity let's assume a and b are synchronized methods.

```
private synchronized void a( ) { b( ); }  
private synchronized void b( ) { a( ); }
```

Java

- Next we have the following scenario:
 1. Thread 1 calls a and is not blocked.
 2. Thread 2 calls b and is not blocked.
 3. Thread 1 attempts to call b but it's blocked.
 4. Thread 2 attempts to call a but it's blocked.
 5. Neither thread will ever be unblocked: they are in deadlock.

In this case the deadlock is easily avoided by introducing a lock with allows exclusive simultaneous access to a and b. Unfortunately, there is no general solution for deadlock prevention.

6 The Chat Application

There are no notes for this section. The lecture slides show a possible implementation.

7 For Wednesday

Study the lecture notes and study Chapter 15.

8 Acknowledgements

Some of this lecture is based on the `Java API` documentation.