Software Development (CS2500)

Lectures 53 & 54: Connections and Threads

M.R.C. van Dongen

February 24 & 26, 2014

Outline

- This lecture is about text-based file I/O and threads.
- We start with a simple `FileReader` object for reading text.
- We continue with studying *buffered* I/O.
- We use a `BufferedFileReader` object to read from a file.
- So far all our `Java` applications have used a single program.
- This lecture studies applications involving *multiple* programs.
- Each program involves a *process* that executes the program.
- Our application lets two programs communicate.
- To do this, we need *threads* on top of our processes.
- As part of this lecture we shall study the `Runnable` interface.

Writing to Text Files

Open: Open the file for writing.

Java

```
FileWriter writer = new FileWriter( ⟨file name⟩ );
```

Write: Zero or more writes.

Java

```
writer.write( ⟨string⟩ );
```

Close: Close the file.

Java

```
writer.close( );
```

Example

Java

```
import java.io.FileWriter;

public class WriteFile {
    public static void main( String[] args ) {
        try {
            FileWriter writer = new FileWriter( "output.txt" );
            writer.write( "My first line of text." );
            writer.write( "My second line of text?" );
            writer.close( );
        } catch( Exception exception ) {
            handleException( exception );
        }
    }
}
```

Example (Continued)

Using a `try`-with-resources Block

Java

```
import java.io.FileWriter;

public class WriteFile {
    public static void main( String[] args ) {
        try ( FileWriter writer = new FileWriter( "output.txt" ) ) {
            writer.write( "My first line of text." );
            writer.write( "My second line of text?" );
        } catch( Exception exception ) {
            handleException( exception );
        }
    }
}
```

Appending to a Text File

- A common operation is appending text to a file.
- Here the file is not overwritten.
- Instead all writes will be added to the end.

Java

```
FileWriter writer = new FileWriter( "MyFile.txt", true );
```

File Objects

- Path names have different representations on different OSs:
  - Differences in path separators,
  - Differences in the root of filesystem,
  - ....
- To overcome these differences, Java defines File type.
- The File class provides abstract file/path names and operations.
- The following are some constructors:

```
File(String parent, String child):
  File(File parent, String child):
            File(String pathname):
```

File Instance Methods

```
boolean canExecute( ):
boolean canRead( ):
boolean canWrite( ):
String getAbsolutePath( ):
String getName( ):
File getParentFile( ):
boolean isDirectory( ):
boolean isFile( ):
boolean isHidden( ):
String[] list( ):
File[] listFiles( ):
boolean mkdir( ):
```

Example

Java

```
public static void main( String[] args ) {
    try {
        File dir  = new File( "Output-Dir" );
        dir.mkdir( );

        File file = new File( dir, "output.txt" );
        FileWriter writer = new FileWriter( file );

        writer.write( "Anybody home?" );
        writer.close( );

        if (dir.isDirectory( )) {
            String path = dir.getAbsolutePath( );
            for (String name : dir.list( )) {
                System.out.println( "Next: " + name );
            }
        }
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

Buffered I/O

- Writing/reading one character at a time is cumbersome:
  - It makes writing your code complicated: loops.
  - It causes overhead.
- *Buffered* reader and writer objects overcome this problem.
- These objects use an internal buffer to optimise I/O.

  Writer:
  - Initially the buffer is empty.
  - Text and characters are written to the buffer.
  - The buffer is *flushed* each time it becomes full.
    - Writes entire buffer to file using one context switch.
  - The buffer is also flushed upon closing.

  Reader:
  - Initially the buffer is filled.
  - Text and characters are read from the buffer.
  - The buffer is filled each time it becomes empty.
- This makes programming read and write operations easier.
- In addition it reduces the number of context switches.

Reading from Text Files

Create `File`

Java

```
public static void main( String[] args ) {
    try ( File file = new File( "input.txt" );
          FileReader fileReader = new FileReader( file );
          BufferedReader reader = new BufferedReader( fileReader ) ) {

        for ( String line = reader.readLine( );
                line != null;
                line = reader.readLine( ) ) {
            System.out.println( "Just read: " + line );
        }
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

Reading from Text Files

Create `FileReader`

Java

```
public static void main( String[] args ) {
    try ( File file = new File( "input.txt" );
          FileReader fileReader = new FileReader( file );
          BufferedReader reader = new BufferedReader( fileReader ) ) {

        for ( String line = reader.readLine( );
                line != null;
                line = reader.readLine( ) ) {
            System.out.println( "Just read: " + line );
        }
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

Reading from Text Files

Create `BufferedFileReader`

Java

```
public static void main( String[] args ) {
    try ( File file = new File( "input.txt" );
          FileReader fileReader = new FileReader( file );
          BufferedReader reader = new BufferedReader( fileReader ) ) {

        for ( String line = reader.readLine( );
                line != null;
                line = reader.readLine( ) ) {
            System.out.println( "Just read: " + line );
        }
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

Reading from Text Files

Minimise Scope of `line`

Java

```
public static void main( String[] args ) {
    try ( File file = new File( "input.txt" );
          FileReader fileReader = new FileReader( file );
          BufferedReader reader = new BufferedReader( fileReader ) ) {

        for ( String line = reader.readLine( );
                line != null;
                line = reader.readLine( ) ) {
            System.out.println( "Just read: " + line );
        }
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

Reading from Text Files

While there's Input

Java

```
public static void main( String[] args ) {
    try ( File file = new File( "input.txt" );
          FileReader fileReader = new FileReader( file );
          BufferedReader reader = new BufferedReader( fileReader ) ) {

        for ( String line = reader.readLine( );
                line != null;
                line = reader.readLine( ) ) {
            System.out.println( "Just read: " + line );
        }
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

Reading from Text Files

Use Input

Java

```
public static void main( String[] args ) {
    try ( File file = new File( "input.txt" );
          FileReader fileReader = new FileReader( file );
          BufferedReader reader = new BufferedReader( fileReader ) ) {

        for ( String line = reader.readLine( );
                line != null;
                line = reader.readLine( ) ) {
            System.out.println( "Just read: " + line );
        }
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

Reading from Text Files

No Need to Close the `BufferedReader`

Java

```
public static void main( String[] args ) {
    try ( File file = new File( "input.txt" );
          FileReader fileReader = new FileReader( file );
          BufferedReader reader = new BufferedReader( fileReader ) ) {

        for ( String line = reader.readLine( );
                line != null;
                line = reader.readLine( ) ) {
            System.out.println( "Just read: " + line );
        }
    } catch( Exception exception ) {
        handleException( exception );
    }
}
```

Making a Connection

- Here we shall implement a chat server application.
  - The server program starts.
  - Chatter programs start connecting to the server.
  - Chatters may send messages to the server.
  - The server broadcasts all messages to its chatters.
  - Upon receiving a message the chatters display the message.
- To get started we shall implement a simple advice server.

An Advice Server

- Our advice server involves a single advisor and a single advisee
- We start by executing the advisor program.
- Next we execute the advisor program.
- The advisee program requests a *connection*.
- The advisor accepts 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.

Connecting, Sending, and Receiving

- The communication depends on server/client machine `IP` addresses.
- Communicating has three ingredients.

  Connect: The client and server establish a `Socket` connection.
  - 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 receives a message.
  - This is done by reading from a `BufferedReader` object.

Making the Socket Connection

- The socket connection is established by creating a `Socket` object.
- Creating the `Socket` formally establishes 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 Java

Java

```
Socket chatsocket = new Socket( ⟨IP address⟩, ⟨port⟩ );
```

- The ⟨IP address⟩ is a String, e.g. "196.164.1.103".
  - It uniquely identifies the server's machine.
- int ⟨port⟩ represents the TCP port on the server's machine.
- The TCP port uniquely identifies some sevice on the server.
  - telnet runs on Port 23,
  - ftp on Port 20, ...
- Valid ports are 0–65535.
- Ports 0–1023 are reserved.

Reading from the Socket

1. Turn the `Socket` into an `InputStream`:

Java

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

2. Turn the `InputStream` into an `InputStreamReader`:

Java

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

3. Turn the `InputStreamReader` into a `BufferedReader`:

Java

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

4. Read:

Java

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

Writing to the Socket

1. Turn the `Socket` into an `OutputStream`:

Java

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

2. Turn the `OutputStream` into a `PrintWriter`:

Java

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

3. Write:

Java

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

4. Flush (if required):

Java

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

The Advisor

Java

```
public static void main( String[] args ) {
    try {
        ServerSocket serverSocket = new ServerSocket( PORT );
        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.
    }
}
```

The Advisee

Java

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

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

    public static void main( String[] args ) {
        try {
            Thread.sleep( 10000 );
            Socket socket = new Socket( IP_ADDRESS, AdviceServer.PORT );
            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.
        }
    }
}
```

Running the Application

Unix Session

```
$
```

Running the Application

Unix Session

```
$ javac Advisee.java AdviceServer.java
```

Running the Application

Unix Session

```
$ javac Advisee.java AdviceServer.java
$
```

Running the Application

Unix Session

```
$ javac Advisee.java AdviceServer.java
$ java AdviceServer &
```

Running the Application

Unix Session

```
$ javac Advisee.java AdviceServer.java
$ java AdviceServer &
[1] 12442
$
```

Running the Application

Unix Session

```
$ javac Advisee.java AdviceServer.java
$ java AdviceServer &
[1] 12442
$ java Advisee &
```

Running the Application

Unix Session

```
$ javac Advisee.java AdviceServer.java
$ java AdviceServer &
[1] 12442
$ java Advisee &
[2] 12456
$
```

Running the Application

Unix Session

```
$ javac Advisee.java AdviceServer.java
$ java AdviceServer &
[1] 12442
$ java Advisee &
[2] 12456
$ ps -a
```

Running the Application

Unix Session

```
$ 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
$
```

Running the Application

Unix Session

```
$ 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
```

Running the Application

Unix Session

```
$ 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
$
```

Our Chat Program

- Our chatter programs do several things:
  - They send messages,
  - They receive messages, and
  - They output their incoming messages.
- How do we implement this?

Don't Try This at Home

```
while (! finished( )) {
    ⟨write message⟩
    ⟨read message⟩
    ⟨display message⟩
}
```

- 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.
- Much better if reading and writing is done simultaneously.
- But how can we do several things at the same time?

What are Threads?

- *Threads* are lightweight processes.
- One `Java` 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.
  - A small stack to enable function calls, and
  - A small area with private data.
- Context switching for threads is faster than for processes.

The Thread Class

Instance Methods

`void run( )` Execute `run( )` in *current* thread.

`void start( )` Start this `Thread` and make it call `run( )`.

The Thread Class

Constructors

`Thread( )`	Create a `Thread` instance.
`Thread( Runnable target )`	Create a `Thread` instance.

The Current `Thread`

Class Method

- `Thread.currentThread( )` returns the current `Thread`.

Creating Threads

Two Ways to Create Them

1. Extend the `Thread` class.
2. Create a class that implements the `Runnable` interface.
   - Create a `Thread` with an instance from that class.

Extending the Thread Class

Java

```
public class Creation extends Thread {
    public static void main( String[] args ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "main = " + current );
        final Thread thread = new Creation( );
        System.out.println( "Running" );
        thread.run( );
        System.out.println( "Starting" );
        thread.start( );
    }

    @Override
    public void run( ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "this = " + this );
        System.out.println( "current = " + current );
    }
}
```

Extending the Thread Class

Java

```
public class Creation extends Thread {
    public static void main( String[] args ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "main = " + current );
        final Thread thread = new Creation( );
        System.out.println( "Running" );
        thread.run( );
        System.out.println( "Starting" );
        thread.start( );
    }

    @Override
    public void run( ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "this = " + this );
        System.out.println( "current = " + current );
    }
}
```

Extending the Thread Class

Java

```
public class Creation extends Thread {
    public static void main( String[] args ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "main = " + current );
        final Thread thread = new Creation( );
        System.out.println( "Running" );
        thread.run( );
        System.out.println( "Starting" );
        thread.start( );
    }

    @Override
    public void run( ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "this = " + this );
        System.out.println( "current = " + current );
    }
}
```

Extending the Thread Class

Java

```
public class Creation extends Thread {
    public static void main( String[] args ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "main = " + current );
        final Thread thread = new Creation( );
        System.out.println( "Running" );
        thread.run( );
        System.out.println( "Starting" );
        thread.start( );
    }

    @Override
    public void run( ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "this = " + this );
        System.out.println( "current = " + current );
    }
}
```

Extending the Thread Class

Java

```
public class Creation extends Thread {
    public static void main( String[] args ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "main = " + current );
        final Thread thread = new Creation( );
        System.out.println( "Running" );
        thread.run( );
        System.out.println( "Starting" );
        thread.start( );
    }

    @Override
    public void run( ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "this = " + this );
        System.out.println( "current = " + current );
    }
}
```

Extending the Thread Class (Continued)

Output

```
main = Thread[main,5,main]
Running
this = Thread[Thread-0,5,main]
current = Thread[main,5,main]
Starting
this = Thread[Thread-0,5,main]
current = Thread[Thread-0,5,main]
```

Extending the Thread Class (Continued)

Output

```
main = Thread[main,5,main]
Running
this = Thread[Thread-0,5,main]
current = Thread[main,5,main]
Starting
this = Thread[Thread-0,5,main]
current = Thread[Thread-0,5,main]
```

Implementing the Runnable Interface

Java

```
public class Creation implements Runnable {
    public static void main( String[] args ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "main = " + current );
        final Thread thread = new Thread( new Creation( ) );
        System.out.println( "Running" );
        thread.run( );
        System.out.println( "Starting" );
        thread.start( );
    }

    @Override
    public void run( ) {
        final Thread current = Thread.currentThread( );
        System.out.println( "this = " + this );
        System.out.println( "current = " + current );
    }
}
```

Implementing the `Runnable` Interface (Continued)

Output

```
main = Thread[main,5,main]
Running
this = Creation@1fe91485
current = Thread[main,5,main]
Starting
this = Creation@1fe91485
current = Thread[Thread-0,5,main]
```

Implementing the Runnable Interface (Continued)

Output

```
main = Thread[main,5,main]
Running
this = Creation@1fe91485
current = Thread[main,5,main]
Starting
this = Creation@1fe91485
current = Thread[Thread-0,5,main]
```

Implementing the `Runnable` Interface (Continued)

Output

```
main = Thread[main,5,main]
Running
this = Creation@1fe91485
current = Thread[main,5,main]
Starting
this = Creation@1fe91485
current = Thread[Thread-0,5,main]
```

Creating Threads

Java

```
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." );
    }
```

Running the Program

Unix Session

```
$
```

Running the Program

Unix Session

```
$ java threadExample
```

Running the Program

Unix Session

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

Race Conditions

- A process has its own private address space.
- A thread doesn't: it shares the *resources* of its process.
- Resources can be: files, variables, and method access.
- Sharing resources violates encapsulation: it may lead to errors.
- To share 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.
  - Program depends on right sequence/timing of other events.
  - Program cannot guarantee right sequence/timing of the events.
- Even read/write operations to/from memory may cause race conditions if they are not properly sequenced.

Race Condition: First Example

Java

```
// 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

Race Condition: First Example

Java

```
// 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	f(1)	0	0	0

Race Condition: First Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1

Race Condition: First Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1

Race Condition: First Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
1	v2 += i	1	1	1

Race Condition: First Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
1	v2 += i	1	1	1
2	f(2)	1	1	1

Race Condition: First Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
1	v2 += i	1	1	1
2	f(2)	1	1	1
2	i = input	1	1	2

Race Condition: First Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
1	v2 += i	1	1	1
2	f(2)	1	1	1
2	i = input	1	1	2
2	v1 += i	3	1	2

Race Condition: First Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
1	v2 += i	1	1	1
2	f(2)	1	1	1
2	i = input	1	1	2
2	v1 += i	3	1	2
2	v2 += i	3	3	2

Race Condition: Second Example

Java

```
// 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

Race Condition: Second Example

Java

```
// 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	f(1)	0	0	0

Race Condition: Second Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1

Race Condition: Second Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1

Race Condition: Second Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
2	f(2)	1	0	1

Race Condition: Second Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
2	f(2)	1	0	1
2	i = input	1	0	2

Race Condition: Second Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
2	f(2)	1	0	1
2	i = input	1	0	2
2	v1 += i	3	0	2

Race Condition: Second Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
2	f(2)	1	0	1
2	i = input	1	0	2
2	v1 += i	3	0	2
2	v2 += i	3	2	2

Race Condition: Second Example

Java

```
// 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	f(1)	0	0	0
1	i = input	0	0	1
1	v1 += i	1	0	1
2	f(2)	1	0	1
2	i = input	1	0	2
2	v1 += i	3	0	2
2	v2 += i	3	2	2
1	v2 += i	3	4	2

Motivation

Dealing with shared resources requires careful synchronisation.

Motivation

Race conditions are caused by lack of synchronisation.

Monitors

- A *monitor* is a high-level synchronisation tool.
  - It is a collection of code that can be executed by no more than one thread at a time.
- In `Java` each `Object` has a *unique* monitor.
  - The `Object`-monitor relationship is one-to-one.
- The programmer defines a monitor by adding code to it.
- `Java` automatically enforces the required synchronisation.
- Guarantees at most one thread has access to the monitor at a time.

Locking

- The monitor may be viewed as a low-level lock.
- A thread can only enter a monitor if the JVM allows it.
- Thread must lock the monitor before it enters the monitor.
  - The thread may only enter the monitor when it is unlocked.
  - When a thread tries entering a locked monitor,
    - The JVM will block that thread.
    - This suspends that thread's execution.
- When a thread leaves the monitor the JVM unlocks the monitor.
  - If threads are blocked, the JVM will release one of them.
  - The released thread may then enter the monitor.
  - This involves (re)locking the monitor.

Object-Monitor Relationship

- Remember that in `Java` each object has its own monitor.
- There are two different kinds of objects:

  Class Objects: Requires locking the class object monitor.
  Normal Objects: Requires locking the object monitor.

synchronized Code

Java

```
synchronized(reference) {
   ⟨body⟩
}
```

synchronized Methods

Java

```
⟨visibility modifier⟩ ⟨class option⟩ synchronized
⟨return type⟩ ⟨name⟩( ⟨argument list⟩ ) {
   ⟨body⟩
}
```

Alternative Notation for `synchronized` Methods

Don't Try This at Home

```
public void instanceMethod( ) {
    synchronized(this) {
        ...
    }
}
```

Eliminating the Race Condition

Java

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

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

Deadlock

- Another problem with concurrent programs is *deadlock*.
- Deadlock happens when several threads 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.

A Deadly Embrace

Shared Resources

Java

```
public class Deadlock {
    int counter = 0;

    public static void main( String[] args ) {
        final Deadlock lock1 = new Deadlock( );
        final Deadlock lock2 = new Deadlock( );
        final Runnable a = new Runnable( ) {
            @Override public void run( ) { grab( lock1, lock2 ); }
        };
        final Runnable b = new Runnable( ) {
            @Override public void run( ) { grab( lock2, lock1 ); }
        };
        new Thread( a ).start( );
        System.out.println( "a running" );
        new Thread( b ).start( );
        System.out.println( "b running" );
    }

    private static void grab( Deadlock first, Deadlock second ) {
        first.counter++;
        sleep( );
        while (second.counter != 0) {
            sleep( );
        }
        first.counter--;
    }
```

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

A Deadly Embrace (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 increments `lock1.counter`.
4. Thread 1 goes to sleep.
5. Thread 2 is created.
6. Thread 2 calls `grab( lock2, lock1 )`.
7. Thread 2 increments `lock2.counter`.
8. Thread 2 goes to sleep.
9. Thread 1 wakes up and notices `lock2.counter != 0`.
10. Thread 1 goes to sleep.
11. Thread 2 wakes up and notices `lock1.counter != 0`.
12. Thread 2 goes to sleep.
13. ....

Deadly Embrace

synchronized Method

Java

```
public class Deadlock {
    public static void main( String[] args ) {
        final Deadlock lock1 = new Deadlock( );
        final Deadlock lock2 = new Deadlock( );
        final Runnable a = new Runnable( ) {
            @Override public void run( ) { grab( lock1, lock2 ); }
        };
        final Runnable b = new Runnable( ) {
            @Override public void run( ) { grab( lock2, lock1 ); }]
        };
        new Thread( a ).start( );
        System.out.println( "a running" );
        new Thread( b ).start( );
        System.out.println( "b running" );
    }

    private static void grab( Deadlock first, Deadlock second ) {
        first.call( second, first );
    }

    private synchronized void call( Deadlock first, Deadlock second ) {
        sleep( );
        if (first != second) {
            first.call( first, first );
        }
    }
```

Deadly Embrace

Shared "Monitor" Resources

Java

```
public class Deadlock {
    public static void main( String[] args ) {
        final Deadlock lock1 = new Deadlock( );
        final Deadlock lock2 = new Deadlock( );
        final Runnable a = new Runnable( ) {
            @Override public void run( ) { grab( lock1, lock2 ); }
        };
        final Runnable b = new Runnable( ) {
            @Override public void run( ) { grab( lock2, lock1 ); }]
        };
        new Thread( a ).start( );
        System.out.println( "a running" );
        new Thread( b ).start( );
        System.out.println( "b running" );
    }

    private static void grab( Deadlock first, Deadlock second ) {
        first.call( second, first );
    }

    private synchronized void call( Deadlock first, Deadlock second ) {
        sleep( );
        if (first != second) {
            first.call( first, first );
        }
    }
```

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

More Deadlock (Continued)

1. Thread 1 is created.
2. Thread 1 calls `grab( lock1, lock2 )`.
3. Thread 1 calls `lock1.call( )`.
4. This locks `lock1`.
5. Thread 1 goes to sleep.
6. Thread 2 is created.
7. Thread 2 calls `grab( lock2, lock1 )`.
8. Thread 2 calls `lock2.call( )`.
9. This locks `lock2`.
10. Thread 2 goes to sleep.
11. Thread 1 wakes up and calls `lock2.call( )`,
    - `lock2` is locked so Thread 1 is blocked.
12. Thread 2 wakes up and calls `lock1.call( )`,
    - `lock1` is locked so Thread 2 is blocked.
13. Both threads are blocked and we're in a deadlock situation.

The Chat Server

Java

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

public class ChatServer {
    public static final int SOCKET = 5000;
    private final ArrayList<PrintWriter> clientOutputStreams;

    public static void main( String[] args ) {
        ChatServer server = new ChatServer( );
        server.serve( );
    }

    private ChatServer( ) {
        clientOutputStreams = new ArrayList<PrintWriter>( );
    }

    private void serve( ) {
        ⟨omitted⟩
    }

    private class ClientHandler implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chat Server

Java

```
private void serve( ) {
    try {
        ServerSocket serverSocket = new ServerSocket( SOCKET );

        while (true) {
            Socket socket = serverSocket.accept( );
            OutputStream os = socket.getOutputStream( );
            PrintWriter writer = new PrintWriter( os );

            synchronized(clientOutputStreams) {
                clientOutputStreams.add( writer );
            }

            ClientHandler handler = new ClientHandler( socket );
            Thread thread = new Thread( handler );
            thread.start( );
            System.out.println( "Made new connection." );
        }
    } catch (Exception exception) {
        exception.printStackTrace( );
    }
}
```

The Chat Server

Java

```
private void serve( ) {
    try {
        ServerSocket serverSocket = new ServerSocket( SOCKET );

        while (true) {
            Socket socket = serverSocket.accept( );
            OutputStream os = socket.getOutputStream( );
            PrintWriter writer = new PrintWriter( os );

            synchronized(clientOutputStreams) {
                clientOutputStreams.add( writer );
            }

            ClientHandler handler = new ClientHandler( socket );
            Thread thread = new Thread( handler );
            thread.start( );
            System.out.println( "Made new connection." );
        }
    } catch (Exception exception) {
        exception.printStackTrace( );
    }
}
```

The Chat Server

Java

```
private void serve( ) {
    try {
        ServerSocket serverSocket = new ServerSocket( SOCKET );

        while (true) {
            Socket socket = serverSocket.accept( );
            OutputStream os = socket.getOutputStream( );
            PrintWriter writer = new PrintWriter( os );

            synchronized(clientOutputStreams) {
                clientOutputStreams.add( writer );
            }

            ClientHandler handler = new ClientHandler( socket );
            Thread thread = new Thread( handler );
            thread.start( );
            System.out.println( "Made new connection." );
        }
    } catch (Exception exception) {
        exception.printStackTrace( );
    }
}
```

The Chat Server

Java

```
private void serve( ) {
    try {
        ServerSocket serverSocket = new ServerSocket( SOCKET );

        while (true) {
            Socket socket = serverSocket.accept( );
            OutputStream os = socket.getOutputStream( );
            PrintWriter writer = new PrintWriter( os );

            synchronized(clientOutputStreams) {
                clientOutputStreams.add( writer );
            }

            ClientHandler handler = new ClientHandler( socket );
            Thread thread = new Thread( handler );
            thread.start( );
            System.out.println( "Made new connection." );
        }
    } catch (Exception exception) {
        exception.printStackTrace( );
    }
}
```

The Client Handler

Java

```
private class ClientHandler implements Runnable {
    private final BufferedReader reader;
    private final Socket socket;

    private ClientHandler( Socket clientSocket ) {
        BufferedReader reader = null;
        socket = clientSocket;
        try {
            InputStream is = clientSocket.getInputStream( );
            InputStreamReader isr = new InputStreamReader( is );
            reader = new BufferedReader( isr );
        } catch ( Exception exception ) {
            // We assume the chatter has left.
        }
        this.reader = reader;
    }

    @Override
    public void run( ) {
        ⟨omitted⟩
    }

    private void broadcast( String message ) {
        ⟨omitted⟩
    }
}
```

The Client Handler

Java

```
@Override
public void run( ) {
    try {
        String string;
        while ((string = reader.readLine( )) != null) {
            broadcast( string );
        }
    } catch( Exception exception ) {
        // We assume the chatter has left.
    }
}

private void broadcast( String message ) {
    System.out.println( "Server received " + message + "" );
    synchronized(clientOutputStreams) {
        for (PrintWriter stream : clientOutputStreams) {
            try {
                stream.println( message );
                stream.flush( );
            } catch (Exception exception) {
                // We assume the chatter has left.
                clientOutputStreams.remove( stream );
            }
        }
    }
}
```

The Chatter

Loopback for `localhost`

Java

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

public class Chatter implements Runnable {
    private static final String IP_ADDRESS = "127.0.1.1";
    private static final int MAX_CHATTERS = 100;
    private static final int MAX_SENDS    = 2;
    private static final int MAX_RECEIVES = 4;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final int id;
    private boolean quit;

    ⟨omitted⟩

    private class Sender implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chatter

Maximum Number of Chatters Allowed

Java

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

public class Chatter implements Runnable {
    private static final String IP_ADDRESS = "127.0.1.1";
    private static final int MAX_CHATTERS = 100;
    private static final int MAX_SENDS    = 2;
    private static final int MAX_RECEIVES = 4;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final int id;
    private boolean quit;

    ⟨omitted⟩

    private class Sender implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chatter

Maximum Sends Per Chatter

Java

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

public class Chatter implements Runnable {
    private static final String IP_ADDRESS = "127.0.1.1";
    private static final int MAX_CHATTERS = 100;
    private static final int MAX_SENDS    = 2;
    private static final int MAX_RECEIVES = 4;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final int id;
    private boolean quit;

    ⟨omitted⟩

    private class Sender implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chatter

Maximum Receives Per Chatter

Java

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

public class Chatter implements Runnable {
    private static final String IP_ADDRESS = "127.0.1.1";
    private static final int MAX_CHATTERS = 100;
    private static final int MAX_SENDS    = 2;
    private static final int MAX_RECEIVES = 4;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final int id;
    private boolean quit;

    ⟨omitted⟩

    private class Sender implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chatter

Each Chatter can Write and Read

Java

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

public class Chatter implements Runnable {
    private static final String IP_ADDRESS = "127.0.1.1";
    private static final int MAX_CHATTERS = 100;
    private static final int MAX_SENDS    = 2;
    private static final int MAX_RECEIVES = 4;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final int id;
    private boolean quit;

    ⟨omitted⟩

    private class Sender implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chatter

Each Chatter has an ID

Java

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

public class Chatter implements Runnable {
    private static final String IP_ADDRESS = "127.0.1.1";
    private static final int MAX_CHATTERS = 100;
    private static final int MAX_SENDS    = 2;
    private static final int MAX_RECEIVES = 4;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final int id;
    private boolean quit;

    ⟨omitted⟩

    private class Sender implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chatter

Chatters Quit if this is Set

Java

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

public class Chatter implements Runnable {
    private static final String IP_ADDRESS = "127.0.1.1";
    private static final int MAX_CHATTERS = 100;
    private static final int MAX_SENDS    = 2;
    private static final int MAX_RECEIVES = 4;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final int id;
    private boolean quit;

    ⟨omitted⟩

    private class Sender implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chatter

Java

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

public class Chatter implements Runnable {
    private static final String IP_ADDRESS = "127.0.1.1";
    private static final int MAX_CHATTERS = 100;
    private static final int MAX_SENDS    = 2;
    private static final int MAX_RECEIVES = 4;
    private final PrintWriter writer;
    private final BufferedReader reader;
    private final int id;
    private boolean quit;

    ⟨omitted⟩

    private class Sender implements Runnable {
        ⟨omitted⟩
    }
}
```

The Chatter

Java

```
public static void main( String[] args ) {
    for (int id = 0; id != MAX_CHATTERS; id++) {
        try {
            Socket socket = new Socket( IP_ADDRESS, ChatServer.SOCKET );
            Runnable chatter = new Chatter( id, socket );
            Thread chatterThread = new Thread( chatter );
            chatterThread.start( );
        } catch( Exception exception ) {
            exception.printStackTrace( );
        }
    }
}
```

The Chatter

Java

```
private Chatter( int id, Socket socket ) throws IOException {
    this.id  = id;
    quit     = false;
    OutputStream os = socket.getOutputStream( );
    writer   = new PrintWriter( os );
    InputStream is = socket.getInputStream( );
    InputStreamReader isr = new InputStreamReader( is );
    reader = new BufferedReader( isr );
}

@Override
public void run( ) {
    Thread receiverThread = new Thread( new Receiver( ) );
    Thread senderThread   = new Thread( new Sender( ) );
    senderThread.start( );
    receiverThread.start( );
}

public synchronized void quit( ) {
    // Modifies shared resource: must be synchronized.
    if (quit) {
        writer.close( );
    } else {
        quit = true;
    }
}
```

The Chatter

Java

```
private class Sender implements Runnable {
    @Override
    public void run( ) {
        try {
            String senderID = "Chatter( Sender ) #" + id;
            int count = 0;
            while (count++ != MAX_SENDS && !quit) {
                Thread.sleep( 1000 );
                String message = "message #" + count;
                writer.println( message + " from " + senderID );
                writer.flush( );
            }
            System.out.println( senderID + " quits" );
            quit( );
        } catch (Exception exception) {
            exception.printStackTrace( );
        }
    }
}
```

The Chatter

Java

```
private class Receiver implements Runnable {
    @Override
    public void run( ) {
        try {
            String receiverID = "Chatter( Receiver ) #" + id;
            int count = 0;
            while (count++ != MAX_RECEIVES && !quit) {
                String message = reader.readLine( );
                String str = receiverID + " receives " + message;
                System.out.println( str );
            }
            System.out.println( receiverID + " quits" );
            quit( );
        } catch (Exception exception) {
            exception.printStackTrace( );
        }
    }
}
```

For Friday

- Study the lecture notes and
- Study [Sierra, and Bates 2004, Chapter 15] (if you have it).

Acknowledgements

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

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.