Software Development (CS2500)

Lectures 53 & 54: Connections and Threads

M.R.C. van Dongen

February 24 & 26, 2014

Contents

1 Outline

This lecture is about text-based file I/O and threads. We start with the I/O part. Here 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.

The second part of this lecture is about threads. The following explains this in a bit more detail. 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. We shall implement an application that lets two programs communicate. As we shall see we shall need several light-weight processes within a program that are more commonly known as *threads*. As part of this lecture we shall study the `Runnable` interface.

2 Writing to Text Files

So far we have read all our input from `System.in` (`stdin`) and we have written all out output from `System.out` (`stdout`). As explained before, `System.in` corresponds to `stdin` and `System.out` corresponds to `stdout`. Both are special cases of unnamed files — in `Java` terminology they are called *streams*. In this section we shall study how to write to a named file.

Writing to a file requires three steps.

Open: Open the file for writing. This involves the creation of a `FileWriter` object. The argument of the constructor is the name of the file.

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

Write: Zero or more writes. Each write appends text to the currently written text. The following demonstrates how to write a string.

```Java
writer.write( ⟨string⟩ );
```

Close: Close the file. Closing the file finishes the writing session. You're not allowed to write to a file that is closed. (Or you have to open it for writing again.)

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

When you create a file, write to a file, or close a files the JVM may throw an exception. You must handle these exceptions. The following example shows how you may handle exceptions in a `try-catch` block. The definition of the method `handleException( )` has been omitted.

```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 );
        }
    }
}
```

When we run this program and look at what's in the file `output.txt` we see that it only contains one line. Newline characters ('\n') characters must be written if you want to create new line:

```Java
writer.write( "My first line of text.\n" );
writer.write( "My second line of text!" );
```

Remember that the `try`-with-resources statement is a much easier way to deal with resources that implement the `AutoClosable` interface.

```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 );
        }
    }
}
```

A common operation is appending text to a file. When you append to a file, `Java` does not overwrite what is already in the file. Instead, `Java` appends each write operation to the end of the file. The following shows how you may create a file in append mode.

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

3 `File` Objects

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

`File(String parent, String child):` Creates a new `File` instance from a parent pathname string and a child pathname string.

`File(File parent, String child):` Creates a new `File` instance from a parent abstract pathname and a child pathname string.

`File(String pathname):` Creates a new `File` instance by converting the given pathname string into an abstract pathname.

`File` objects provide a wide range of useful methods. The following are but a few:

`boolean canExecute( ):` Tests whether the application can execute the file denoted by this abstract pathname.

`boolean canRead( ):` Tests whether the application can read the file denoted by this abstract pathname.

`boolean canWrite( ):` Tests whether the application can write the file denoted by this abstract pathname.

`String getAbsolutePath( ):` Returns the absolute pathname string of this abstract pathname.

`String getName( ):` Returns the name of the file or directory denoted by this abstract pathname.

`File getParentFile( ):` Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.

`boolean isDirectory( ):` Tests whether the file denoted by this abstract pathname is a directory.

`boolean isFile( ):` Tests whether the file denoted by this abstract pathname is a normal file.

`boolean isHidden( ):` Tests whether the file named by this abstract pathname is a hidden file.

`String[] list( ):` Returns an array of pathnames denoting the files in the directory denoted by this abstract pathname.

`File[] listFiles( ):` Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.

`boolean mkdir( ):` Creates the directory named by this abstract pathname.

The following is a small example which demonstrates some of the functionality. Notice that in the previous section we used a `FileWriter` constructor that took a `String` argument. The following uses a constructor with a `File` argument.

```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 );
    }
}
```

4 Buffered I/O

Writing/reading one character at a time is cumbersome:

- It makes writing your code complicated: loops.
- It causes overhead. Each read/write corresponds to a method call, which is relatively costly in terms of time. The following explains this in more details.

  For security reasons modern CPUs have different protection levels/modes for different kinds of operations. Common protection levels are *user mode* for normal operations and *protected* or *operation system mode* for operations that require more security. Read and write operations are sensitive operations that are only out in protected mode. If the processor is in user mode and it requires a read or write operation, the processor requires a *context switch* from user mode to protected mode. After the context switch to protected mode, the processor carries out the read or write operation and then carries out another context switch to user mode so the JVM can continue executing the program. A context switches is a time-expensive operation, so character-based IO, which requires two context switches per character, causes a lot of overhead.

Buffered reader and writer objects overcome this problem. These objects use an internal buffer to optimise I/O. How the buffer is used depends on the kind object.

Writer: For writer objects this works as follows. Initially the buffer is empty. Text and characters are written to the buffer. The buffer is *flushed* each time it becomes full. Here *flushing* the buffer means writing the entire buffer content to the file. This only requires one context switch. The buffer is also flushed when the file object is closed.

Reader: For reader objects this works similar but "the other way around". 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.

5 Reading from Text Files

Reading from text files works similar as writing to text files. The following example shows how to read some lines of text from a named file. Again we use the `try`-with-resources statement to initialise the `reader` instance.

```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 );
    }
}
```

Most of this is pretty straightforward. Notice that we use a `for` statement to read the input. We *could* also have read the input as follows.

```Java
String line;
while ( (line = reader.readLine( )) != null ) {
    System.out.println( "Just read: " + line );
}
```

Arguably, the previous technique is clearer. but it has a flaw because the scope of the variable `line` now reaches beyond the `while` statement. With the `for` construct the scope of the variable is minimised, thereby preventing lots of errors that are caused such as typos, other editing operations, and others.

6 Making Connections

In this part of the lecture we shall implement a simple chat server application. First the server program starts. Next the chatter programs start connecting to the server. Chatters may send messages to the server. When this happens the server broadcasts all messages to its chatters. Upon receiving a message the chatters display the message. Before we start with the chat server, we shall carry out a case study that involves a simple advice server.

Our advice server application involves a single advisor and a single advisee. program. Both programs execute *at the same time.* 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.

6.1 Bird's Eye View

The server and client communication depends on the the server's `IP` address. Communicating has three ingredients.

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 receives 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.

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

```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. The ⟨`port`⟩ is an `int` representing the `TCP` port on the server's machine. The `TCP` port uniquely identifies some sevice on the server. For example, `telnet` runs on Port 23, `ftp` on Port 20, ... Valid ports are 0–65535. Ports 0–1023 are reserved.

6.3 Reading from the Socket

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

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( );
   ```

6.4 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`:

```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. The `PrintWriter` is a *buffered* writer. Sine the `PrintWriter` is buffered, its write operations are forwarded to its buffer before sending them to the target destination. Normally, the `PritWriter` only sends its buffer contents to the target destination when its buffer is full. This may mean that the advisee has to wait a long time before it gets the advice server's advice. Clearly, we should make sure the advisee receives the advice sooner, which is why we *flush* the buffer. Basically, flushing the buffer sends the current buffer contents to the destination and then empties the buffer.

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

6.5 The Advisor

Remember that our application involves two programs: an advice server program, and an advisee program. The following is the `AdviceServer` class. For sake of this example, the program generates random advice using a class method called `giveAdvice( )`. The `main( )` is presented further on.

```Java
import java.util.Random;
import java.io.*;
import java.net.*;
public class AdviceServer {
    public static final int PORT = 5000;
    private static final Random rand = new Random( );
    private static final String[] advices = { "Go for it.", "Don't." };

    // main( ) omitted.

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

The following is the `main( )`.

```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 server starts by creating a `ServerSocket`, which waits for incoming requests. 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 must use the same number.

Next the server starts an infinite loop that listens to the socket.[1] The call to `serverSocket.accept( )` *blocks* until a client requests a connection. To say that a call *blocks* on some condition means that the call doesn't return until 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.

6.6 The Advisee

The following is the `Advisee` class.

```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.
        }
    }
}
```

[1] A proper implementation should have a proper termination mechanism. However, for the purpose of this lecture we use an infinite loop because implementing the termination only complicates the example.

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.

When the client returns from the `sleep( )` call it continues by creating its `Socket` by connecting to `AdviceServer.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.

6.7 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
$
```

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.

7 Threads

7.1 Motivation

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

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. It would be much better if we could do the reading and writing at the same time. But how can we do several things at the same time? In the remainder of this lecture we shall learn how `Java` threads let us carry out several things at the same time. When we know enough about threads, we shall implement our chat server application.

7.2 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. 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.

7.3 The `Thread` Class

The following are some of the instance methods defined in the `Thread` class.

`void run( )` This method calls `run( )` in the current thread. It works just like any other instance method call, so it carries out the statements of the method `run( )` and then returns.

Note that when we write *thread,* we mean lightweight process; when we write *`Thread`,* we mean object. The *current* thread is the lightweight process that executes the current bytecode statements. The `Thread` class has a class method called `Thread.currentThread( )` that returns a `Thread` object reference representing the current thread (lightweight process).

`void start( )` This *starts* a new thread. When this happens, the JVM will (1) create a new thread (lightweight process), (2) execute the new thread along with the current thread, and (3) let the current thread return from the call to `start( )`.

Note that `start( )` is an instance method, so the method is called by/using a `Thread` object that owns the method: the method call is executed by the current thread. Usually, we call this `Thread` the "`this`." When the JVM creates the new thread, the new thread is created using *`this`* `Thread`. When the JVM starts executing the new thread, it forces the thread to execute the `run( )` method that is owned by `this Thread`. When the thread returns from the call to `run( )` it terminates.

The following are some of the constructors defined in the `Thread` class.

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

`Thread( Runnable target )` Create a `Thread` instance. The resulting `Thread` will call `target.run( )` when its own `run( )` method is called.

The class method `Thread.currentThread( )` returns the current `Thread`, i.e. the thread that currently executes the `Java` program.

7.4 Creating Threads

There are (at least) two techniques to create a new `Thread`.

1. Extend the `Thread` class.
2. Create a class that implements the `Runnable` interface. Next you create a `Thread` with an instance from the resulting class.

The following shows the first of the two techniques: extend 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 );
    }
}
```

When you execute this program your output should be along the following lines.[2]

```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]
```

To understand why calling `run( )` and `start( )` results in different output, remember that calling `thread.run( )` is executed in the *current* thread of execution. This is the same as the one that executes the `main( )` (and the first line of output). When you call `start( )`, the JVM creates a new thread of execution and lets the resulting thread execute the call `run( )`. This explains why calling the `run( )` method outputs two different `Strings` for `this` and `current` and why calling `start( )` outputs the same `Strings`.

You may also create a `Thread` by creating a `Thread` instance from an instance of a class that implements the `Runnable` interface. This is the second and last technique to create a `Thread` instance. The following shows this technique.

[2] The output may differ because of differences in the JVM.

```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 );
    }
}
```

When you execute this program your output should be along the following lines.

```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]
```

This time the `this` in the method `run( )` corresponds to an instance of the `Creation` class, which should explain the difference between this example and the previous example.

In the following example, we use the second technique and create two threads, which are then started. This causes the resulting threads to run *concurrently*.

```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." );
    }
}
```

Even if you hadn't seen the `Runnable` interface before, you should be able to tell, just by looking at the example, that the `Runnable` interface defines an abstract method `public void run( )`. Without the `@Override` notation you may not have been able to tell this. This once more shows how important/useful it is you use the notation: it's a form of documentation that helps others understand your code.

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

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

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` that 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 ….

Question 1. *What would happen with the previous program if you used two calls to `run( )` instead of two calls to `start( )`?*

```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

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 that 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.

8 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 share their resources properly 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.
- The program *itself* cannot guarantee the right sequence and/or timing of the 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.)

8.1 Example

Figures 1 and 2 demonstrate how a race condition may occur. In both figures there are two threads that 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 that should seemingly hold is broken at the end of the sequence of events 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.

```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

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 that 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.

8.2 Monitors

`Java` has an easy solution to prevent (some) race conditions. This section briefly explains the easiest way to prevent them. The solution involves the notion of *monitors,* which are a synchronisation tool that helps us enforce mutual exclusive access to some code segments. In the following, we shall first study monitors, how to define monitors with the keyword `synchronized`, and how to avoid race conditions with monitors. In the next lectures we shall study monitors in more detail. We shall also study other mechanisms for avoiding race conditions.

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. The `Java` language automatically enforces the required synchronisation. This guarantees that *at most one thread has access to the monitor at a time.*

8.3 Locking

The monitor may be viewed as a low-level lock. A thread can only enter a monitor if the JVM allows it. When a thread enters the monitor the JVM locks 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. Effectively, that thread's execution is temporarilly suspended and the JVM puts the thread—a `Thread` object—is in the object's *monitor queue.* When a thread leaves the monitor the JVM unlocks the monitor. If threads are blocked on the monitor, the JVM will remove one of them from the monitor queue. The released thread may then enter the monitor,which involves (re)locking the monitor.

8.4 Object-Monitor Relationship

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

Class Objects: In Java each class is represented by a unique class object. When a thread enters a monitor that is owned by a class object, this requires locking the monitor that is owned by the class object. A thread can only enter a class object monitor if it calls a class method that belongs to the monitor or (but this is rare) when it starts executing other statements that belong to the monitor.

Normal Objects: When a thread enters a monitor that is ownwed by a normal (non-class) object, this requires locking the object's monitor. A thread can only enter a normal object monitor if it calls an instance method that belongs to the monitor or when it starts executing statements that belong to the monitor.

8.5 Defining Monitors

Now that we know about monitors, we're ready to learn how to define them. Remember that in Java each monitor is associated with a unique object. Likewise each object has its own (unique) monitor. The keyword synchronized lets you add code to object monitors. There are two ways you can do this.

The following is the first way you can use the synchronized keyword.

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

In this variant the ⟨body⟩ inside the block is added to the monitor of the object reference reference. There are two possibilities.

1. If the code is in a class method, the code is added to that class's monitor
2. If the code is in an instance method, the code is added to the current instance's monitor

The following is the second way you can use the keyword synchronized.

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

In this variant the entire method is added to the object monitor. There are two possibilities. If the method is a class method, the method is added to the class's object monitor. If the method is an instance method, the method is added to the instance's monitor.

With this notation we can also implement a synchronized instance method as follows. However, this should only be taken as an example. Marking the entire method as synchronized is much clearer.

```
public void instanceMethod( ) {
    synchronized(this) {
        ...
    }
}
```
Don't Try this at Home

```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 */
}
```

Figure 3: Multi-threaded program. Since the method `f( )` is `synchronized` it is guaranteed that the invariant is maintained provided (1) the invariant holds when the method is entered and (2) all access to the variables `i`, `v1`, and `v1` is done in `synchronized` methods.

8.6 Eliminating the Race Condition

Using `synchronized` methods we can fix the implementation of Figure 2. All we have to do is making the method `f( ) synchronized`. This is shown in Figure 3.

9 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. Notice that each thread needs the shared resources called `lock1` and `lock2`. However, this will turn out impossible because each will grab one of the resources and cannot grab the other resource because the other thread isn't willing to release its resource. The method call `sleep( )` lets the current `Thread` sleep for 1 millisecond.

```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--;
    }
}
```

Next we have the following scenario:

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. ....

The following example is similar. Here we can have deadlock too because two threads compete for getting simultaneous access to the monitor of two objects. One way of viewing this example is that the threads compete for access to the monitors. A different way is viewing the example is that the threads compete for the objects that own the monitors (because the object-monitor relationship is one-to-one). Again the definition of `sleep( )` is omitted.

```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 );
        }
    }
}
```

Next we have the following scenario:

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.

Unfortunately, there is no general solution for deadlock prevention. However, for simple cases there may be solutions.

10 The Chat Application

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

11 For Friday

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

12 Acknowledgements

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