6SENG006W Concurrent Programming

Week 5

Introduction to Java Threads

PART I

Overview of Java Concurrency

Overview of Java Concurrency

In our examination of Java's concurrency features we shall look at three different levels:

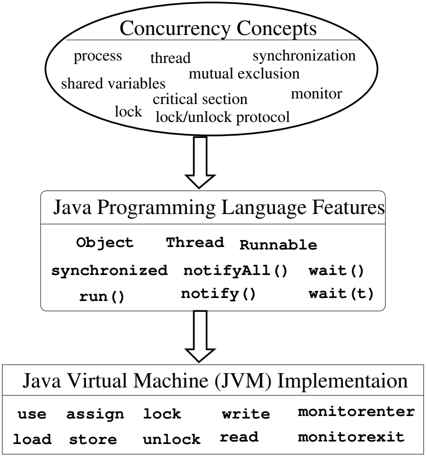

Figure : 5.1 Java concurrency

Concurrent Programming in Java

Concurrent programming in Java is achieved through the use of multiple *“threads” of execution* in programs.

The programs which use multiple threads are called *multi-threaded* programs.

These programs require special handling, since they usually *share data* & therefore involve *synchronisation* to **avoid interference**.

The main aspects of Java threads we shall cover are:

- the *definition* of a thread,
- the `Thread` class & the `Runnable` interface,
- how threads are represented in the *Java Virtual Machine* (JVM).
- the *life-cycle* of a thread,
- thread *scheduling*,
- *thread groups* & the `ThreadGroup` class,
- thread *synchronisation*,
- Java *Semaphores*,
- Java *Monitors*.

The Java development environment supports threaded programs through the language, libraries & run time system.

PART II

Introduction to Threads

Introduction to Threads

The aim of this lecture is to introduce *Java Threads* by:

- explaining how *concurrent programming* is achieved in Java;
- introducing the notion of *processes* & *threads*;
- describing *thread attributes*;
- explaining how to *create a thread* using the two available approaches:
  - by *sub-classing* the `Thread` class; or
  - by *implementing* the `Runnable` interface;
- presenting two simple examples of multi-threaded programs:
  - `TwoThreadsTest` (& `SimpleThread`) that subclass the `Thread` class;
  - `Clock` applet that implements the `Runnable` interface;
- presenting a brief overview of how threads are represented in the *Java Virtual Machine* (JVM).

Java Processes & Threads

In concurrent programming, there are two basic units of execution: *processes* & *threads*.

Java concurrent programming is mostly concerned with threads, but processes are also important.

A computer system normally has many active processes & threads.

This is true even in systems that only have a *single processor* (or *execution core)* & thus only have one thread *actually executing at any given moment*.

Processing time for a single core is shared among processes and threads through an operating system feature called *time slicing*.

It is now very common for computer systems to have multiple processors or processors with multiple execution cores, thus these can support true concurrent execution of processes & threads.

But concurrency is possible even on simple systems, without multiple processors or execution cores.

What is a Process?

A process has a *self-contained execution environment*.

A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications.

However, what the user sees as a single application may in fact be a set of *cooperating processes*.

To facilitate communication between processes, most operating systems support *Inter Process Communication (IPC)* resources, such as *pipes* & *sockets*.

IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the JVM run as a single operating system process.

What is a Thread?

Threads are sometimes called *lightweight processes*.

Both processes & threads provide an *execution environment*.

But creating a new thread requires fewer resources than creating a new process.

Threads *exist within a process* – every process has at least one.

Threads *share* the process's resources, including *memory* & *open files*.

This can result in the efficient use of resources & communication between the threads.

But it also leads to the introduction of the associated problems that arise with concurrency, e.g. **interference**, **deadlock**, etc. etc.

Multi-threaded Execution & the Java Platform

Multi-threaded execution is an essential feature of the Java platform.

Every application has at least one thread.

In practice it also usually has several *"system"* threads that do things like memory management (e.g. garabge collection) & signal handling.

But from the application programmer's point of view, you start with just one thread, called the *"`main`"* thread.

The `main` thread has the ability to create additional threads, as we shall see.

More about Threads

A *sequential program*: has a beginning, an end, a sequence, & at any given time during the run time of the program there is a *single point of execution*.

A single *thread* is similar to a sequential program.

However, a thread itself is **not a program** – it cannot run on its own – but runs within a program.

> **Definition:** *Thread*
>
> A *thread* is a *single sequential flow of control* within a process.

A single thread is not very interesting, but multiple threads in a single program all running at the same time & performing different tasks is.

A thread is similar to a real process in that a thread & a running program are both a *single sequential flow of control*.

Threads are “Lightweight”

For the reasons just outlined, a thread is considered *“lightweight”*.

This is because a *thread*:

- runs within the context of a full-blown program &
- takes advantage of the resources allocated for that program &
- accesses the program’s environment
- but must have its own resources within a running program, e.g.
  - its own *execution stack*,
  - its own *program counter*.

The code running in the thread works only within that context.

Thread synonyms are: *execution context* & *lightweight process*.

A Simple Thread Example

`SimpleThread` ***creates*** & ***starts*** two independent ***threads***.

```
class SimpleThread extends Thread
{
  final int OneSecond = 1000 ;

  public SimpleThread( String str )
  {
    super( str ) ;      // ``Thread( String )'' constructor
  }

  public void run()     // ``body'' of the thread
  {
    for ( int i = 0 ; i < 10 ; i++ )
    {
      System.out.println( getName() + ": " + i ) ;
      try {
            sleep( (int)( Math.random() * OneSecond ) ) ;
          }
      catch ( InterruptedException e ) {}
    }
    System.out.println( getName() + ": TERMINATING" ) ;
  }

} // SimpleThread
```

TwoThreadsTest the Main Class

```
class TwoThreadsTest    // the ``main'' class (program)
{

   public static void main ( String args )
   {
     // Declare 2 thread variables
     Thread firstThrd ;
     Thread secondThrd ;

     // Create the 2 threads
     firstThrd  = new SimpleThread( "FirstThread" ) ;

     secondThrd = new SimpleThread( "SecondThread" ) ;

     // Start the 2 threads executing
     firstThrd.start() ;

     secondThrd.start() ;
   }

} // TwoThreadsTest
```

Diagrammatic view of the program

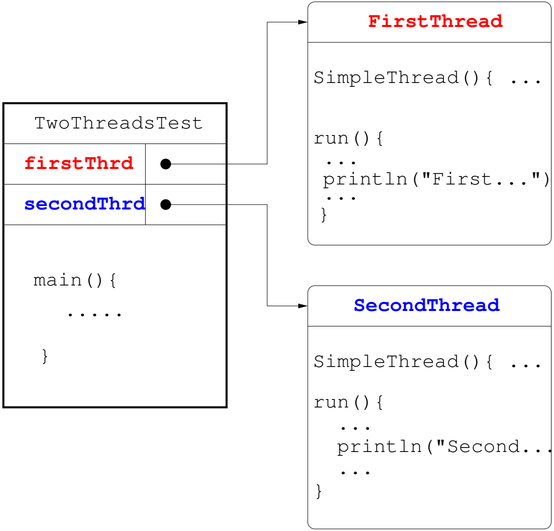

Figure : 5.2 `TwoThreadsTest` program

`SimpleThread` Class: Subclassing & Constructor

The `SimpleThread` class is a *subclass* of the `Thread` class that is provided by the `java.lang` package.

The first method in the `SimpleThread` class is a *constructor* that takes a `String` as its only argument.

This constructor is implemented by calling a super class constructor & sets the thread's name which is used later in the program.

NOTE: when sub-classing the `Thread` class, the constructors of the subclass:

MUST ALWAYS CALL ONE OF THE `Thread` CLASS'S CONSTRUCTORS.

The full details of the *9* `Thread` class *constructors* can be found at:

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Thread.html

`SimpleThread` Class: `run()` Method

The next method in the class is the `run()` method.

The `run()` method is the heart of any thread & where the action of the thread takes place, i.e. it is the *"main"* method for a thread.

The `run()` method of the `SimpleThread` class contains a `for` loop that iterates ten times.

In each iteration the `run()` method displays the iteration number & the name of the thread using the `Thread` class's method: `getName()`.

It then sleeps for a random interval between 0 & 1 seconds.

After the loop has finished, the `run()` method prints the thread's name followed by "`TERMINATING`" then terminates.

The `TwoThreadsTest` Class

The `TwoThreadsTest` class's `main` method:

- *declares* two `Thread` variables named: "`firstThrd`" & "`secondThrd`".
- *creates two* `SimpleThread` threads named: "`FirstThread`" & "`SecondThread`".
- *starts the execution* of each thread following its construction by calling the `Thread` class's `start()` method.

NOTE:

YOU NEVER EVER CALL A THREAD'S `run()` METHOD!

YOU ONLY EVER CALL THE THREAD CLASS'S `start()` METHOD!

One Possible Output from `TwoThreadsTest`

```
FirstThread: 0
SecondThread: 0
SecondThread: 1
FirstThread: 1
FirstThread: 2
SecondThread: 2
SecondThread: 3
FirstThread: 3
FirstThread: 4
SecondThread: 4
FirstThread: 5
SecondThread: 5
SecondThread: 6
FirstThread: 6
FirstThread: 7
SecondThread: 7
SecondThread: 8
SecondThread: 9
FirstThread: 8
SecondThread: TERMINATING
FirstThread: 9
FirstThread: TERMINATING
```

The output from the 2 `SimpleThread` threads is *interleaved*, because their `run()` methods are running *"concurrently"* – each thread is displaying its output at the same time as the other.

Exercise: Add a third thread with the name "`ThirdThread`".

Summary & Conclusions

- The important idea is that a Java program can have *many threads*, & that those threads run *concurrently*.
- In reality, the threads may be executed using *real concurrency* or only using *pseudo-concurrency*.
- The important point is not what method is used, but that the programmer treats the threads *as if they are executed concurrently*.
- In this way **no assumptions** will be made about the *order* in which the threads: *start*, *execute statements* or *terminate*.
- Consequently, *fewer errors* are likely to be made when designing & programming the threads.

PART III

What You Need to Know About Threads – "Attributes"

What You Need to Know About Threads

Java threads are implemented by the `java.lang.Thread` class.

The `Thread` class implements a *system independent* definition of Java threads which is specified by the *Thread API*.

To use threads efficiently & without errors you must understand the following aspects of threads & the Java run time system:

- how to *create* a thread,
- how to provide a *body* for a thread,
- the *life-cycle* of a thread, i.e. *"thread states"*,
- how the run time system *schedules* threads, i.e. *"thread priority"*,
- what *daemon* threads are & how to write them,
- how to use *thread groups*,
- how to *coordinate* their activities using *synchronisation*.

Key Thread “Attributes” (I)

Creating a Thread

A thread can be created in *two* ways:

1. by *sub-classing* the `Thread` class & *overriding* its `run()` method; or

2. creating a Thread with “a `Runnable` object as its *target*”, i.e. use an object that implements the `Runnable` interface.

Thread Body

All of the action takes place in the thread’s body – that is the thread’s `run()` method.
How the thread’s body – `run()` method is defined is related to how the thread is created.

Thread State

Throughout its life, a Java thread is in *one of several states*.
A *thread’s state* indicates what the thread is doing & what it is capable of doing at that time of its life: *running*, *not running*, *waiting*, *sleeping* or *dead*.

Key Thread “Attributes” (II)

Thread Priority

A thread’s *priority* indicates to the Java thread scheduler when this thread *should run* in relation to all of the other threads.

Daemon Threads

Daemon threads are those that provide a *service* for other threads in the system.

Any Java thread can be a daemon thread.

Thread Group

All threads belong to a *“thread group”*.

`ThreadGroup`, is a `java.lang` class, which defines and implements the capabilities of a group of related threads.

PART IV

Two Ways to Create a Thread

Creating a Thread & the `run()` method

There are *two* ways to create a thread & define its associated `run()` method:

`Thread` Class

Subclass the `Thread` class defined in the `java.lang` package & *override* the `run()` method, e.g. the `SimpleThread` class.

`Runnable` Interface

Provide a class that *implements* the `Runnable` interface, i.e. provides an *implementation* of the `run()` method.
The `Runnable` interface is also defined in the `java.lang` package.

Which method is chosen will depend on the context but the following guideline provides assistance.

Guideline: If your class *must* subclass some other class (e.g. `Applet`), you should use `Runnable` as described in 2.

For full details of the `Runnable` interface see:

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Runnable.html

Thread Body – `run()`

After a thread has been *created* & *initialised*, the *run time system calls* its `run()` method.

The `run()` method implements the *“behaviour”* of the thread.

Often, a thread’s `run()` method is a loop, e.g. an animation thread might loop through & display a series of images.

Sometimes a thread’s `run()` method performs an operation that takes a long time, e.g. down loading & displaying an image.

There are *two* ways that you can provide a customised `run()` method for a Java thread.

We have seen how to do this by subclass the `Thread` class in the `TwoThreadsTest` & `SimpleThread` example.

We shall now look at an example of how to do this by implementing the `Runnable` interface.

The `Runnable` Interface

The `Runnable` interface is defined in the `java.lang` package:

```
public interface Runnable
{
        public abstract void run() ;
}
```

It defines a single method called `run()` that does not accept any arguments & does not return a value.

When a class *implements* the `Runnable` interface it must provide an *implementation* for the `run()` method as defined in the interface.

Warning: If it does **not** then the implementing class will **not be able to be instantiated**, as it will be an `abstract` class.

Steps for Creating a Thread Using the `Runnable` Interface

1. *Define* a class that *implements* the `Runnable` interface, i.e. provides an *implementation* of the `run()` method.

2. *Instantiate* this class, i.e. create an instance of it (an object).
   This object is known as the *"runnable object"* or *"runnable target"*, e.g. `runnableobject`.
   This will provide the `run()` method for the new thread.

3. *Create* the new thread using a *constructor* for either:
   - the `Thread` class, e.g.
     `Thread thrd = new Thread( runnableobject ) ;`
   - a *subclass* of the `Thread` class, e.g.
     `Thread thrd = new SubClassOfThread( runnableobject ) ;`

   The *constructor* **must be** passed a *reference* to an instance of a class (object) that implements the `Runnable` class, e.g. `runnableobject`.

4. Finally, *start the thread*: `thrd.start()`

Template for Creating a Thread Using the `Runnable` Interface

Here `SubClass` ***subclasses*** `SuperClass` ($\neq$ `Thread`), but wants to use a thread, so it does so by ***implementing*** the `Runnable` interface.

```
public class SubClass extends SuperClass implements Runnable
{
   private Thread utilityThread ;

   public SubClass()              // Constructor
   {
     // ......

     // Create the thread using "this" as the "runnable target"
     utilityThread = new Thread( this, "utilityThread" ) ;

     utilityThread.start() ;      // Start the utilityThread

    // ......
   }

   public void run()              // ``run()'' ONLY used by utilityThread
   {
      // body for the utilityThread
   }

   // Define other SubClass methods

} // SubClass
```

Diagrammatic View of `SubClass`

Fig. 5.3 shows the relationship between the *classes* & *interface* in the `SubClass` program.

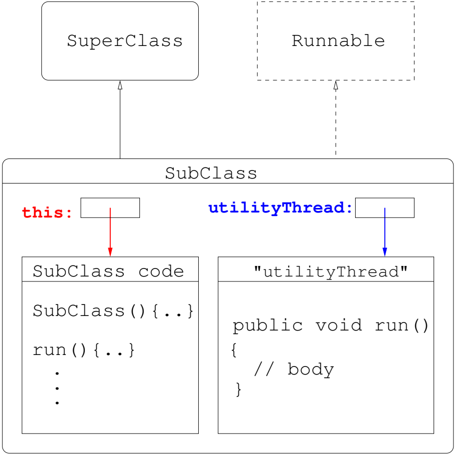

Figure : 5.3 `SubClass` program

Example: The `Clock` Applet

`Clock` applet displays the current time & updates its display every second.

It does this by creating a *"utility thread"* that continuously prints out the time & then sleeps for one second.

```
import java.applet.Applet ;
import java.awt.Graphics  ;
import java.util.Calendar ;

public class Clock extends Applet implements Runnable
{
  private Thread clockThread = null ;

  public void start()                // Applet's "start()" method
  {
    if ( clockThread == null )
    {
       clockThread = new Thread(this, "Clock") ;
       clockThread.start() ;       // Thread's "start()" method
    }
  }
```

Notes: `Clock` *"implements"* the `Runnable`, so its a *"runnable target"*;
& in "`Thread(this, "Clock")`", "**`this`**" represents the `Clock` applet.

`Clock` Example Code Continued

```
// implementing ``run'' makes Clock a ``runnable target''
public void run()
{
  while ( clockThread != null ) {
        repaint();
        try {
               clockThread.sleep( 1000 ) ;
        } catch (InterruptedException e){ }
  }
}

public void paint(Graphics g){
  Calendar now = Calendar.getInstance();

  g.drawString( now.get( Calendar.HOUR_OF_DAY ) + ":" +
                now.get( Calendar.MINUTE )      + ":" +
                now.get( Calendar.SECOND ), 5, 10  ) ;
}

public void stop(){       // Applet's "stop" method
    clockThread = null ;
}

} // Clock
```

Note: setting **`clockThread = null`** will *eventually stop* the `Clock` thread, i.e. after `sleep(1000)` returns, to kill it more quickly `interrupt` it.

Using the `Clock` Applet

The applet could then be called by loading the following web page into either the `appletviewer` program or using a browser, e.g. *Firefox*.

```
<HTML>
  <HEAD>
    <TITLE> The Clock Applet </TITLE>
  </HEAD>
  <BODY>
    Here is the output of the clock:
    <APPLET CODE = "Clock.class"
            WIDTH = 150
            HEIGHT = 25>
    </APPLET>
  </BODY>
</HTML>
```

Diagrammatic View of `Clock`

Fig. 5.4 is an overview of the threads involved in the `Clock` applet.

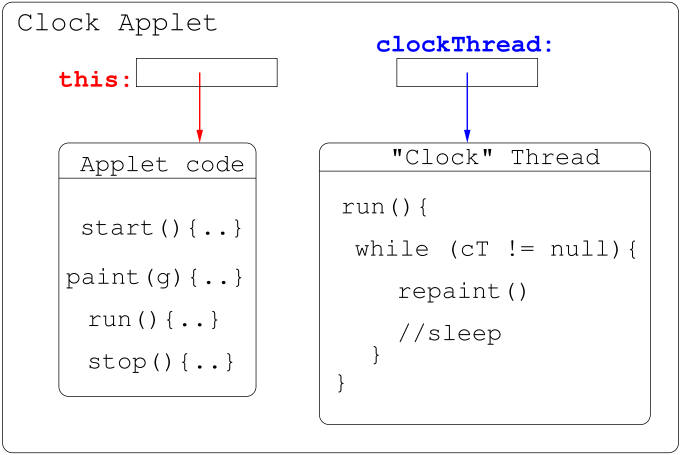

Figure : 5.4 `Clock` applet

How the Clock Applet Creates its Thread

The class definition for the `Clock` class indicates that the `Clock` is a *subclass* of `Applet` & *implements* the `Runnable` interface.

```
class Clock extends Applet implements Runnable
```

The `Clock` applet *creates its thread* – **`clockThread`** by:

- Creating **`clockThread`** using one of the `Thread` class's constructors that takes a *"runnable object"* as an argument.
- Then because the `Clock` class *implements* the `Runnable` interface, it **must provide an implementation for the `run()` method** as defined in the interface.
  This ensures that an instance of the `Clock` class is a *"runnable object"*.
- The `Clock`'s **`run()`** method is then used as the *body of the thread* **`clockThread`**.
  The **`run()`** method is **NEVER** called directly by any method within the `Clock` applet.

How it is Executed

The application in which an applet is running calls the applet's `start()` method when it loads the applet.

The Clock applet creates a thread named **`clockThread`** in its `start()` method & starts the thread.

First, the `start()` method checks to see if **`clockThread`** is `null`.

If **`clockThread`** is `null`, the applet is *new* or has been *previously stopped* & a *new thread must be created*; otherwise, the applet is *already running*.

The applet *creates a new thread* by calling a `Thread` class constructor:

```
clockThread = new Thread(this, "Clock") ;
```

Where: "**`this`**" represents the `Clock` applet & "`Clock`" is the thread's name.

The first argument to this `Thread` constructor **must implement the `Runnable` interface** & becomes the thread's *"target"*.

When constructed in this way, the thread (**`clockThread`**) gets its **`run()`** method from its target `Runnable` object, i.e. the `Clock` applet.

Stopping the `clockThread` Thread

When the page is left the application in which the applet is running calls the applet's `stop()` method.

The `Clock`'s `stop()` method *stops the thread* by:

```
clockThread = null ;
```

this stops the continual updating of the clock, by terminating the `while`-loop in the **`run()`** method.

In earlier versions of JDK (1.1.5/7) the thread could be stopped by using the `Thread` class's **`stop()`** method:

```
clockThread.stop() ;
```

which would stop it immediately.

The problem with this method was that the thread could be stopped at an *"inappropriate point"* which **could result in errors**.

For this reason the **`stop()`** method has been *"deprecated"* since JDK 1.2.

If the page is revisited, the `start()` method is called again, and the clock starts up again with a *new thread*.

The `run()` Method

The `Clock`'s **`run()`** method provides the heart of the `Clock` applet.

When the applet is asked to **stop**, the applet stops the **`clockThread`** by setting it to **`null`**; this lets the **`run()`** method know when to stop.

The first line of the **`run()`** method loops until **`clockThread`** is **`null`**.

Within the loop, the applet repaints itself & then tells the thread to sleep for 1 second (1000 milliseconds).

The applet's `repaint()` method calls the `paint()` method which does the actual update of the applet's display area.

The `Clock`'s `paint()` method gets the current time & draws it to the screen.

Questions:
What happens if the web page is returned to within 1 second of leaving it?

Will the **`clockThread`** have realized that it should die?

Could another **`clockThread`** be created?

How could you change the `while`-loop condition to take account of this?

Deciding to Use the `Runnable` Interface

The `Clock` class is *derived from* the `Applet` class, so that it can run in a Java-compatible browser,

But, the `Clock` applet also needs to use a thread so that it can continuously update its display without taking over the process in which it is running.

Java does **not support multiple-inheritance**, therefore the `Clock` class could **not inherit from both** the `Thread` & `Applet` classes.

```
public class Clock extends Applet, Thread // NOT ALLOWED
```

So, the `Clock` class uses the `Runnable` interface to provide the **`run()`** method for its threaded behaviour.

Applets are **not threads** nor do Java-compatible browsers or applet viewers automatically create threads in which to run applets.

Therefore, if an applet needs any threads it must create its own.

The `Clock` applet needs one thread to perform its frequent display updates & the user needs to be able to perform other tasks at the same time.

PART V

Threads & the Java Virtual Machine (JVM)

Threads & the Java Virtual Machine (JVM)

Overview

The JVM can support *many threads of execution at once*.

These threads *independently execute code* that operates on values & objects residing in a *shared main memory*.

Threads may be supported by: *many hardware processors*; *time-slicing a single hardware processor*; *time-slicing many hardware processors*.

Outline the relationship between *threads* & the *JVM* by:

- presenting an example of a *multi-threaded program* in the JVM;
- describing how a thread *accesses a shared variable*;
- describing the *interaction* of threads with the *main memory*, by means of *low-level* actions;

(For more details on how Java threads are implemented in the JVM see the references on the module web site.)

Multi-Threaded programs in the JVM

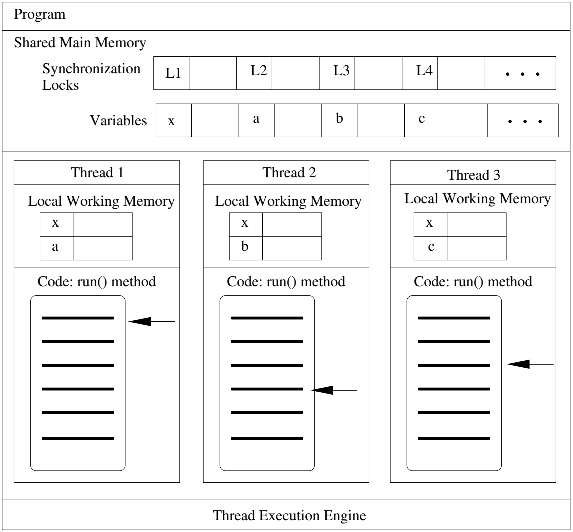

Figure : 5.5 Three Threads in the JVM

Description of 3 Threads Program in the JVM

Consider the above program with 3 threads active in the JVM.

The 3 threads use *4 variables*:

- `x` is *shared by all threads*,
- `a`, `b`, `c` are **not shared** between the threads.
- Each of `a`, `b`, `c` is used by only one thread.

Each of the three threads has its own:

- *local memory*,
- *current instruction pointer*,
- in general they will be at *different stages in their execution*.

How Threads Communicate & Execute

- *Communicate by means of shared variables*, e.g. `x`.
- *Variables* are kept in *main memory* & are *shared* by all threads.
- Main memory contains the *master copy* of every variable.
- Every thread has its own local *working memory* where it keeps its own *working copy* of variables, e.g. `a`, `b`, `c`.
- Use *local copies* of variables & transfer them to & from *main memory*.
- Threads are executed by a *Thread Execution Engine*.
- As a thread executes a program, it operates on these working copies, i.e. it *uses* them or *assigns* to them.
- It is **impossible** for one thread to access *parameters* or *local variables* of another thread.
- Main memory contains *locks*.
- There is *one lock associated with each object*.
- Threads may **compete** to *acquire* a lock, but will eventually *release* it.

How a Thread Accesses a (Shared) Variable

As a thread executes code, it carries out a sequence of *actions*, that may *use* (read) the value of a variable or *assign* (write) it a new value.

If two or more concurrent threads act on a *shared variable*, there is a possibility that the actions on the variable will **produce timing-dependent results**.

This dependence on timing is *inherent in concurrent programming* & means that the result of executing a concurrent program **can be unpredictable**.

There are *rules* about when a thread is *permitted* or *required* to transfer the contents of its *working copy* of a variable into the *master copy* or vice versa.

Rules for Transferring Variables between: Working Memory & Main Memory

These *rules* ensure that:

1. To access a shared variable, a thread usually first obtains a *lock* & flushes its working memory.
2. This guarantees that shared values will thereafter be loaded from the shared main memory to the threads working memory.
3. When a thread unlocks a lock it guarantees the values it holds in its working memory will be written back to the main memory.

Thread Interaction with the JVM's Main Memory

Figure 4.6 shows the *interactions* of a thread with the *main memory*, using the JVM's *low-level actions* when a thread executes the assignment: "`a = x ;`".

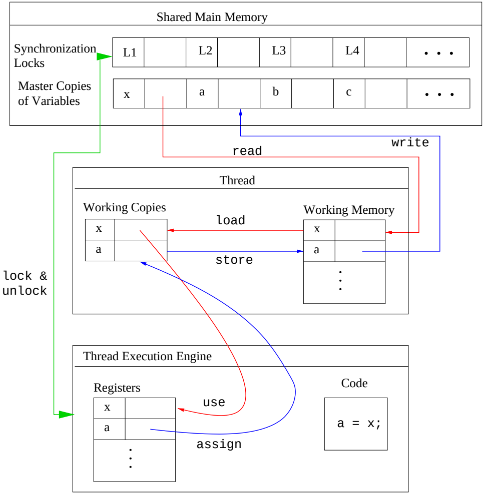

Figure : 5.6 Threads & Main Memory Interaction in the JVM

Interaction of Threads with the Main Memory (I)

The *interaction of threads with the main memory*, & thus with each other, is in terms of the following *low-level actions*:

Thread actions:

- `use` transfers the contents of a variable in the thread's *working copy* memory to the thread's *execution engine*.
  Performed whenever a thread uses the value of a variable.
- `assign` transfers a value from the thread's *execution engine* into the thread's *working copy* of a variable.
  Performed whenever a thread assigns to a variable.
- `load` puts a value transmitted from *main memory* by a `read` action into the thread's *working copy* of a variable.
- `store` transmits the contents of the thread's *working copy* of a variable to *main memory* for use by a later `write` operation.

Interaction of Threads with the Main Memory (II)

Main Memory actions:

- `read` transmits the contents of the *master copy* of a variable to a thread's *working memory* for use by a later `load` operation.
- `write` puts a value transmitted from the thread's *working memory* by a `store` action into the *master copy* of a variable in *main memory*.

Thread & Main Memory "Tightly Synchronized" actions:

- `lock` causes a thread to **acquire one claim on a lock**.
- `unlock` causes a thread to **release one claim on a lock**.

Constraints

Each of these operations is *atomic*, i.e. *indivisible*.

A `use` or `assign` operation is a *tightly coupled interaction* between a thread's execution engine & its working memory.

A `lock` or `unlock` operation is a *tightly coupled interaction* between a thread's execution engine & the main memory.

The transfer of data between the main memory & a thread's working memory is *loosely coupled*.

The *main memory* performs a:

- `read` operation for every `load`, &
- a `write` operation for every `store`.