Programming with threads

Topics

1. Introduction
2. Threads in Java
3. Concurrent performance
4. Communication between threads
5. Synchronization
6. Monitors and condition synchronization
7. Semaphores
8. Properties of activity
9. Design and implementation of thread-safe classes.

Additional literature

[Ca] Campione & Walrath: The Java Tutorial. Second Edition. Object-Oriented Programming for the Internet. Add.-Wesl. 1999. 0-201-31007-4.
(Has been published in a 3. edition:0-201-70393-9).
CD-version: Trail: Essential Java Classes: Doing Two or More Tasks at Once: Threads.

[De] Deitel: Java. How to Program. Sixth Edition. Chapter 23. Multithreading.
Pearson Education. 2005.

1 Introduction

1.1 Parallel system

A *parallel system* shall *handle separate activities*, which are ongoing concurrently *(at the same time/simultaneously).*

Two activities are parallel *if they at a given time are ongoing somewhere between their starting point and ending point.*

In some cases the separate activities will
cooperate
on solving a given problem.

In other cases the separate activities will
compete
about joint system resources which they have to share.

In order to get an activity executed, a program module, which can be carried out on a processor, is required.

Generally a *process* can be characterised as an *active object*, which carry out the activity on a processor with the program module.

A *parallel system* will therefore consist of program modules which are being executed concurrently by several processes (at one or several processors).

1.2 Processes and threads

In many operating systems and languages you distinguish between *processes* and *threads*.

The *process* term is the "heavy" term.
An important purpose of *processes* is to *protect their adjoining programs from each other*, as the processes cannot in general be assumed to be cooperative – rather the contrary.

A *process* can therefore be perceived as a *"shell", that surrounds and protects an activity*, - typically the performance of an application. To the process one can tie a number of resourcer, which are thus protected from misuse by other processes. Processes are carried out in parallel. The *activity in one process is handled by one or several threads.*

As an important part of the protection of its activity the process is awarded

a separate address room.

In some literature a *thread* is also called a *lightweight process*.

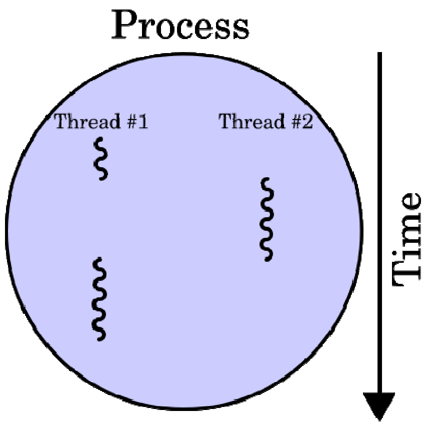


Figure 1. A process with two threads

In many connections, such as control and regulation, one need (lightweight) processes which must be carried out simultaneously (parallel), and often *have to cooperate*. Such (lightweight) processes shall therefore *not be protected from each other*.

A *thread* (lightweight process) is an *independent sequential activity within a process.*

A process can contain several parallel *threads*, which share the process resources and are *not protected from each other.*

Thus threads have

shared address room, i.e. the process address room.

This address room includes among others

global variables and the heap.

Furthermore each thread has

its own separate stack.

This is shown in the below figure which illustrates a process with three threads.

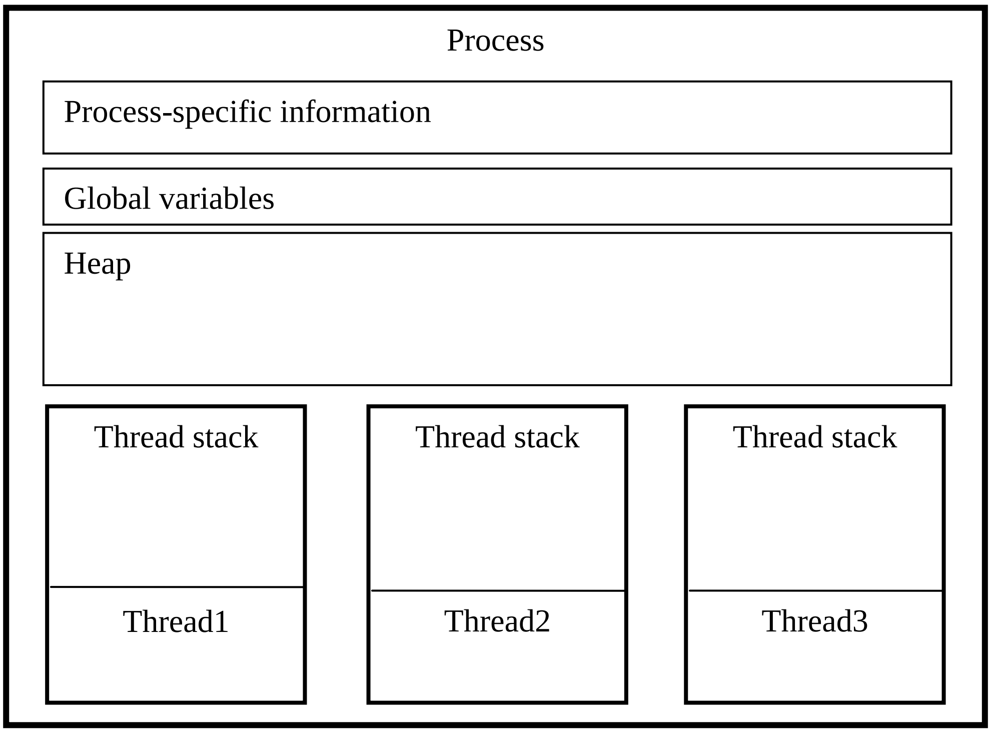


Figure 2. A process with three threads

2 Threads in Java

Also see [Ca].

In the Java package `java.lang` you can find

```
public interface Runnable
public class Thread extends Object implements Runnable
```

`public interface Runnable`

The `Runnable` *interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called* `run`. *This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example,* `Runnable` *is implemented by* `class Thread`. *Being active simply means that a thread has been started and has not yet been stopped.*

`public class Thread extends Object implements Runnable`

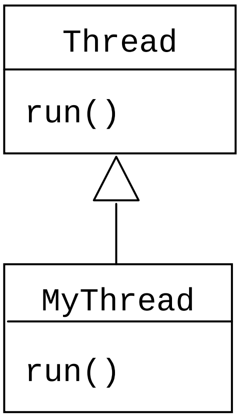
A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

There are two ways to create a new thread of execution.

2.1 Thread in the first way

One is to declare a class to be a subclass of `Thread`. *This subclass should override the* `run` *method of* `class Thread`. *An instance of the subclass can then be allocated and started.*

`class Thread` carries out instruction from its method `run()`. The actual code which is being carried out depends on the implementation of `run()` in a derived class which appear from the following structural diagram:



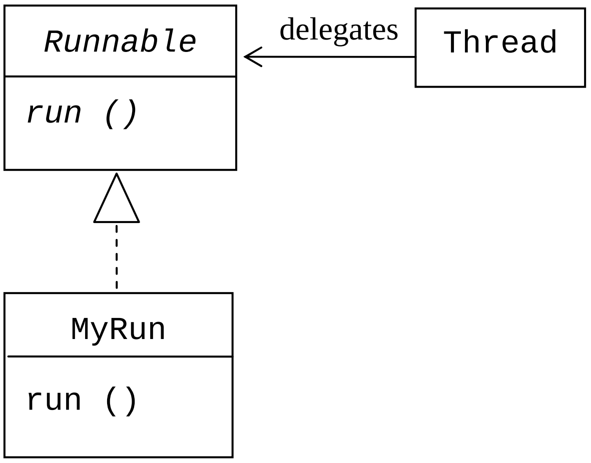
```
class Thread implements Runnable
{
  public void run () {..}
  public void start(){..}
}

class MyThread extends Thread
{
  public void run ()
  {
    // to be implemented
  }
}
```

Figure 3. Implementation of `run()` by inheritance.

2.2 Thread in the second way

The other way to create a thread is to declare a class that implements the `Runnable` *interface. That class then implements the* `run` *method. An instance of the class can then be allocated, passed as an argument when creating* `Thread`, *and started.*

As Java does not have multiple inheritance it is some times necessary to implement the `run()`-method in a class derived from `interface Runnable`.



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

class MyRun implements Runnable
{
  public void run ()
  {
    // ..
  }
}
```

Figure 4 Implementation of `run()` by `interface Runnable`.

Notice that class `MyRun` is not a thread class. E.g. it has no `start` method.

2.3 Creation of and running a thread object

In the two cases a thread object has to be created, after which the thread can be started.

In the first case it is carried out as follows:

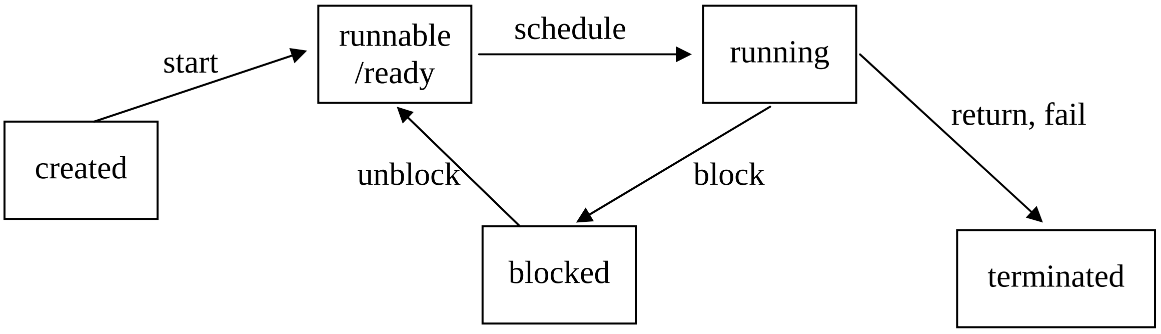
```
Thread th1 = new MyThread ();
th1.start ();
```

In the second case a thread object shall also be created but this thread delegates the running to an object of type `Runnable`, which has an implemented `run` method. In our case it is `MyRun`-object. It is carried out as follows:

```
Thread th2 = new Thread (new MyRun ());
th2.start ();
```

Notice, that the `run` method shall not be called in any of the two cases. This is carried out inside the `start` method.

2.4 Life cycle of a thread



Also see [Ca].

2.5 The methods of `class Thread`

See the Java-documentation. Some few methods are shown here:

```
void start()
void run()
static void sleep(long millis)
boolean isAlive()
public final void join()          waits for the thread to die
static void yield()               causes thread to pause
```

Furthermore some methods which *you must not use (<u>deprecated</u>)*:

```
void stop()
void resume()
void suspend()
```

When a company like Sun (and without doubt also many other companies) can make such grave mistakes, which mistakes might we, who are only mortals, then make? Among others it *<u>shows how careful you must be when you make parallel programs with threads that communicate mutually</u>*. Also see the Java-documentation for e.g. `stop()`, which refers to:

> *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?*
>
> *Why is Thread.stop deprecated? Because it is <u>inherently unsafe</u>.*
>
> *Why are Thread.suspend and Thread.resume deprecated? Thread.suspend is <u>inherently deadlock-prone</u>.*

Sun has been criticized from numerous parties, among others by Per Brinch Hansen, who argues that its handling of communication between threads is on a too "low level".

(Per Brinch Hansen: Java's Insecure Parallelism, 1999. Can be found on the Net.)

3 Concurrent execution

3.1 Programs with several threads that use the same object (shared object)

Example 3.1.1: A thread which updates a counter.

```
class Counter
{
  private long value = 0;

  public void inc () {
    value++;
  }

  public long value () {
    return value;
  }
}

class Thread1 extends Thread
{
  private Counter counter;
  private final int N = 200*1000;

  public Thread0 (String name, Counter counter )
  {
    super (name);
    this.counter = counter;
  }

  public void run () {
    for (int in = 0; i < N; i++)
      counter.inc ();
    System.out.println (getName () + " finished: Counter.value = " +
                        counter.value());
  }
}

public class TestThreadEks2
{
  public static void main (String[] args)
  {
    Counter counter = new Counter ();
    Thread t01 = new Thread0 ("Thread01", counter);
    t01.start ();
  }
}
```

Example 3.1.2: Two threads which update the *same* counter.

The above program is extended so that `main` contains 2 threads that use the same counter:

```
Counter counter = new Counter ();
Thread t01 = new Thread0 ("Thread01", counter);
Thread t02 = new Thread0 ("Thread02", counter);

t01.start ();
t02.start ();
```

What does the program print? Which value does the counter have at the end?

3.2 Race Condition

The above example shows a situation where two treads read and write to the same variable, namely the `counter`'s `value`, and where the final result is not predictable. The result depends on how the two threads incidentally are working.

Such a situation is called *race condition*, because the result depends on which thread that wins the race of access to the shared variable.

3.3 Race Condition in Example 1.1

To find out why race condition occurs when threads increment the same counter, we will look at how the `inc`-method is translated to Java bytecode.

We use the `javap` command in a command prompt:

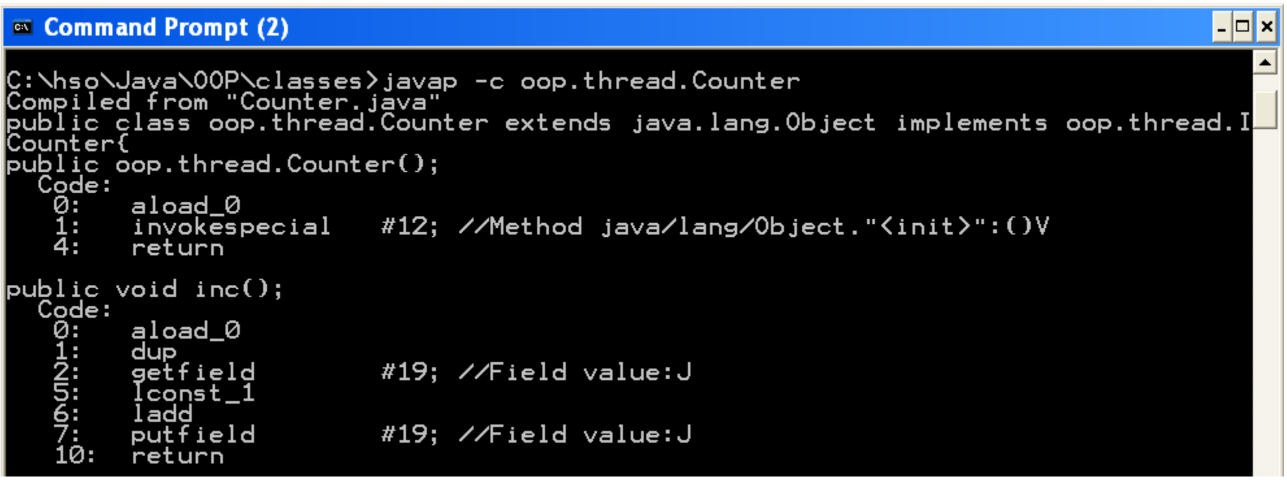
```
C:\hso\Java\OOP\classes
javap –c oop.thread.Counter
```

javap - The Java Class File Disassembler

The **javap** command disassembles a class file.

Option **-c**

> Prints out disassembled code, i.e., the instructions that comprise the Java bytecodes, for each of the methods in the class.



```
C:\hso\Java\OOP\classes>javap -c oop.thread.Counter
Compiled from "Counter.java"
public class oop.thread.Counter extends java.lang.Object implements oop.thread.I
Counter{
public oop.thread.Counter();
  Code:
   0:   aload_0
   1:   invokespecial   #12; //Method java/lang/Object."<init>":()V
   4:   return

public void inc();
  Code:
   0:   aload_0
   1:   dup
   2:   getfield        #19; //Field value:J
   5:   lconst_1
   6:   ladd
   7:   putfield        #19; //Field value:J
   10:  return
```

The disassembled `inc` code is translated to seven bytecode instructions.

Knowing this, it should be possible to explain why race condition occurs in the example.