**Threads in the JVM**

The JVM distinguishes between daemon and non-daemon threads.

* Non-daemon – include the main thread, the Event Dispatch Thread, and your own created threads.
* Daemon threads are used for various JVM execution tasks such as garbage collection.

The system cannot terminate while there are non-daemon thread being executed but will exit even if there are daemon threads still performing tasks.

**Getting started with threads**

The Thread class.

There are two important methods in this class:

* void start()
* void run()

A simple thing to remember about these is that **you should never call method run() from your code**.

The run() method contains the code which the Thread object will execute. This can of course make calls out to other methods of other classes (think of it as the main method for this thread). The run() method must be invoked by the JVM using the Thread object that is going to run it.

The start() method is an instruction that we give the JVM to being execution of the thread that we call it on. This will cause the JVM to invoke run().

**Creating a thread**

We can create a thread by writing:

...

Thread somethread = new Thread();

...

And to start it:

...

somethread.start();

...

This tells the JVM to create an actual thread, give it an id, create a Java Stack, get it ready for scheduling, and invoke method run() with it by pushing a frame for run() on to this new thread’s stack.

**Making a thread do something**

In order to make a thread actually perform some code we need to provide some code in the run() method. An obvious way of doing this is to write a subclass of Thread which overrides the run() method.

class MyThread extends Thread {

public void run(){

System.out.println("I'm a thread that does something");

}

}

public class MyProg {

public static void main(String[] args){

Thread t = new MyThread();

t.start();

System.out.println("And I do something too!");

}

}

**Interface runnable**

Interface Runnable contains a single method: void run(). Therefore to implement this interface we simply need to implement the run() method.

To use an object which implements this interface as a Thread we can use the constructor method in Thread which accepts a Runnable object.

class MyThread implements Runnable {

public void run(){

System.out.println("I'm a Runnable thread that does something");

}

}

public class MyProg {

public static void main(String[] args){

Runnable r = new MyThread();

Thread t = new Thread(r);

t.start();

System.out.println("And I do something too!");

}

}

//Replace first three lines of the main method

//new Thread( new MyThread() ).start();

**So what can I do with a thread?**

We can use the sleep command.

public static void main(String[] args){

System.out.println("Started, hang on a second.");

try {

Thread.sleep(1000);

}

catch (InterruptedException e){ return; }

System.out.println("A second has elapsed.");

}

Or another example

public class SleepTest {

public static void main(String[] args){

new Thread (new SleepyThread(5, args[0])).start();

new Thread (new SleepyThread(1, args[1])).start();

System.out.println("The main thread dies here");

}

}

class SleepyThread implements Runnable {

int st;

String msg;

public SleepyThread(int st, String msg){

this.st = st; this.msg = msg;

}

public void run(){

try { Thread.sleep(st\*1000); } catch (InterruptedException e){ }

while(true){

try { Thread.sleep(500); } catch (InterruptedException e){ }

System.out.print(msg+" ");

}

}

}

**Thread priorities**

The Java threading model uses a system of priority level to allow certain thread to execute before other, less important, threads. Each thread is assigned an integer priority level between 1 and 10. The lower the number, the less important the thread is.

The priority level are used as hints to the thread scheduler to indicate which threads should be run first. Threads which have the same priority level are supposed to be executed in a round robin fashion (they take turns executing for a fixed amount of time until every thread has a turn – then they being all over again).

A Thread which has been newly created inherits the same priority as the thread which called start() to create it. The default priority value is 5. There are three constants:

* Thread.MAX\_PRIORITY
* Thread.MIN\_PRIORITY
* Thread.NORM\_PRIORITY

**Interrupting threads**

try { ... } catch(InterruptedException e)

One thing you can do to disturb the execution of another thread is to ‘interrupt’ it. You do this by calling interrupt() on the thread that you wish to disturb.

What this actually does, provided the calling thread has permission to do so, is it sets the interrupted flag in the receiver thread to be true. If you have programmed the receiver thread to observe interruptions, then the control flow can be altered this way:

class Interruptee extends Thread {

public void run(){

while (true){

while (! this.isInterrupted()){

System.out.print(" Zzzz ");

try { Thread.sleep(1000); }

catch (InterruptedException e)

{ System.out.println("You woke me up"); }

}

System.out.print(" Ouch ");

}

}

}

public class Interruptor {

public static void main(String[] args){

java.util.Random r = new java.util.Random();

Thread it = new Interruptee() ;

it.start();

for (int i=0; i<5; i++){

try { Thread.sleep(r.nextInt(5)\*1000); }

catch (InterruptedException e) { }

it.interrupt();

}

}

}

The call to interrupt from the main thread sets the interrupt flag in the Interruptee to be true – unless this thread is sleeping, in which case an InterruptedException is thrown instead. To clear the interrupt flag you can call the static method interrupted() which is very similar isInterrupted but it cats on the executing thread (rather than the receiver of the method call) and it clears the flag after inspecting it.

**Variable concurrency**

All Threads share the heap. Which means that any fields which are declared in an object may be accessible to a number of different threads at any one time.

Race condition – two or more threads compete for access to modify a shared resource.

Join method – when we invoke the join() method on a thread, the calling thread goes into a waiting state. It remains in a waiting state until the referenced thread terminates.