CS360 Lecture 10
Multithreading

Tuesday, March 9, 2004

Reading

Exception Handling: Chapter 15
Multithreading: Chapter 16

Declaring New Exception Types

Most programmers use existing exception classes from the Java API or from the third party vendors.

If you do need to create an exception class, then you should extend the class Exception and specify two constructors.

```
public class MyException extends Exception
{
    public MyException()
    {
    }

    public MyException(String reason)
    {
        super(reason);
    }
}

public class NonZero
{

    public NonZero(int number)
        throws MyException
    {
        if( number == 0 )
            throw new MyException(
                      "The argument was zero");

        System.out.println("Number " + number
                              + " is non zero");
    }

    public static void main(String[] args)
    {
        try
        {
            NonZero nz1 = new NonZero(1);
            NonZero nz0 = new NonZero(0);
            System.out.println("All OK");
        }
        catch( MyException e )
        {
            System.out.println(e);
        }
    }
}
```

Multithreading

Performing operations concurrently on computers means that we can perform two operations at the same time.

Question: Can computers execute operations concurrently?

Historically, concurrency has been implemented as operating-system primitives available only to experienced system programmers. Then Ada came along. It supported concurrency primitives.

Java makes concurrency primitives available to application programmers. Applications contain threads of execution, where each thread designates a portion of a program that may execute concurrently with other threads.

Question: When would an application need to use multithreading?

Threading mechanisms in various operating systems handle thread scheduling differently.

Issues with Concurrent Programming

Writing concurrent programs presents issues that do not occur when writing sequential code:

- Safety: Two different threads could write to the same memory location at the same time leaving the memory location in an improper state
- Liveness: Threads can become deadlocked, each thread waiting forever for the other to perform a task.
- Nondeterminism: Thread activities can become intertwined. This can make the program harder to debug.
- Communication: Different threads in the same program execute autonomously from each other. Communication between threads is an issue.

States of a Thread
Start, run, etc.

Example

```
public class ExtendingThreadExample extends Thread
{
  public void run()
  {
    for( int count = 0; count < 4; count ++ )
      System.out.println( "Message " + count +
                          " From: Mum" );
  }

  public static void main( String args[] )
  {
    ExtendingThreadExample parallel = new
                               ExtendingThreadExample();
    System.out.println( "Create the thread" );
    parallel.start();
    System.out.println( "Started the thread" );
    System.out.println( "End" );
  }
}
```

Output

```
Create the thread
Started the thread
End
Message 0 From: Mum
Message 1 From: Mum
Message 2 From: Mum
Message 3 From: Mum
```

Output

```
Create the thread
Message 0 From: Mum
Message 1 From: Mum
Message 2 From: Mum
Message 3 From: Mum
Started the thread
End
```

Giving a Thread a Name

Each thread can be given its own name, which can be useful.

```
public class ExtendingThreadExample extends Thread
{
  public ExtendingThreadExample( String name )
  {
    super( name );
  }

  public void run()
  {
    for( int count = 0; count < 4; count ++ )
      System.out.println( "Message " + count + " From: " +
                          Thread.currentThread().getName() );
  }

  public static void main( String args[] )
  {
    ExtendingThreadExample parallel = new
                            ExtendingThreadExample( "Dad" );
    System.out.println( "Create the thread" );
    parallel.start();
    System.out.println( "Started the thread" );
    System.out.println( "End" );
  }
}
```

The next example shows a simple thread that we will be using throughout the lecture.

```
public class SimpleThread extends Thread
{
  private int maxCount = 32;

  public SimpleThread( String name )
  {
    super( name );
  }

  public SimpleThread( String name, int repetitions )
  {
    super( name );
    maxCount = repetitions;
  }

  public SimpleThread( int repetitions )
  {
    maxCount = repetitions;
  }

  public void run()
  {
    for( int count = 0; count < maxCount; count++ )
    {
      System.out.println( count + " From: " + getName() );
    }
  }
}
```

The following example uses the SimpleThread class. The output depends on whether a multiprocessor or single processor machine is used to run the program.

```
public class RunSimpleThread
{
  public static void main( String args[] )
  {
    SimpleThread first = new SimpleThread( 5 );
    SimpleThread second = new SimpleThread( 5 );
    first.start();
    System.out.println( "Started " + first.getName() );
    second.start();
    System.out.println( "Started " + second.getName() );
    System.out.println( "End" );
  }
}
```

Output

```
0 From: Thread-1
1 From: Thread-1
2 From: Thread-1
3 From: Thread-1
4 From: Thread-1
Started Thread-1
0 From: Thread-2
1 From: Thread-2
2 From: Thread-2
3 From: Thread-2
4 From: Thread-2
Started Thread-2
End
```

Thread Priority

Each thread has a priority. If there are two or more active threads and one has a higher priority than others, the higher priority thread is run until it is done or not active.

java.lang.Thread field	Value
Thread.MAX_PRIORITY	10

Thread.NORM_PRIORITY	5
Thread.MIN_PRIORITY	1

How to Determine Priorities

Continuously running parts of the program should have lower priority than rarer events.

User inputs should have very high priority.

A thread that continually updates some data should have the lowest priority.

```
public class PriorityExample
{
  public static void main( String args[] )
  {
    SimpleThread first = new SimpleThread( 5 );
    SimpleThread second = new SimpleThread( 5 );
    second.setPriority( 10 );
    first.setPriority( 1 );
    first.start();
    second.start();
    System.out.println( "End" );
  }
}
```

```
0 From: Thread-2
1 From: Thread-2
2 From: Thread-2
3 From: Thread-2
4 From: Thread-2
End
0 From: Thread-1
1 From: Thread-1
2 From: Thread-1
3 From: Thread-1
4 From: Thread-1
```

Threads only run once. Once a thread has been run it can’t be restarted.

Thread Scheduling

Time-slicing

A thread is run for a short time slice and suspended. It resumes only when it gets its next turn. Threads of the same priority share turns.

Non time-sliced

Threads run until:

- they end
- they are terminated
- they are interrupted, higher priority threads interrupts lower priority threads
- they go to sleep
- they block on some call
- they yield

Java does not specify if threads are time-sliced or not. Implementations are free to decide.

```
public class InfinityThread extends Thread
{
  public void run()
  {
    while( true )
      System.out.println( "From: " + getName() );
  }

  public static void main( String args[] )
  {
    InfinityThread first = new InfinityThread();
    InfinityThread second = new InfinityThread();
    first.start();
    second.start();
  }
}
```

What do you think is output?

Types of Threads

Several of the examples shown continue running even after the main method has completed running.

The reason for this is because they are user threads. In Java there are two types of threads:

- Daemon threads are expendable. When all the user threads have completed, the program ends and all daemon threads are stopped
- User threads are not expendable. They continue to execute until their run method ends or an exception propagates beyond the run method.

```
public class DaemonExample extends Thread
{
  public static void main( String args[] )
  {
    DaemonExample shortLived = new DaemonExample();
    shortLived.setDaemon( true );
    shortLived.start();
    System.out.println( "Bye" );
  }

  public void run()
  {
    while( true )
    {
      System.out.println( "From: " + getName() );
      System.out.flush();
    }
  }
}
```